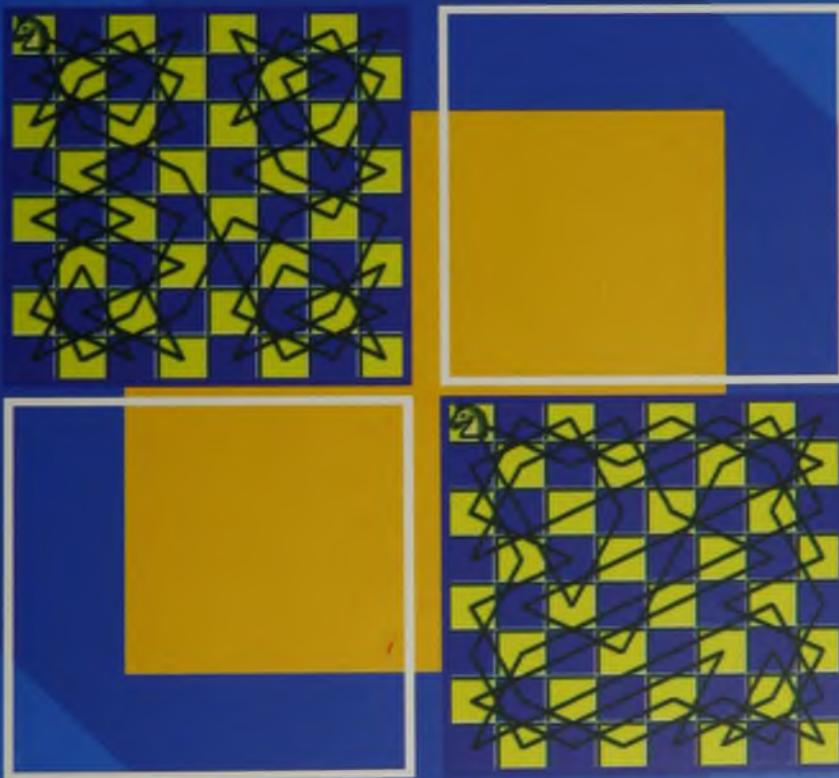


TRẦN THÔNG QUẾ

# CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

(PHÂN TÍCH VÀ CÀI ĐẶT TRÊN C/C<sup>++</sup>)

(Tập 1)



NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

TRẦN THÔNG QUẾ

CẤU TRÚC  
DỮ LIỆU VÀ THUẬT TOÁN

(PHÂN TÍCH VÀ CÀI ĐẶT TRÊN C/C<sup>++</sup>)

(Tập 1)

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

**Mã số: HT 04 HM14**

## LỜI NHÀ XUẤT BẢN

Trong thời đại bùng nổ công nghệ thông tin như hiện nay, dữ liệu đang được sinh ra ngày càng nhiều và dưới nhiều hình thức khác nhau. Việc xử lý nhằm ứng dụng kho dữ liệu quý giá đó như thế nào để đạt được hiệu quả cao nhất luôn là vấn đề được rất nhiều những chuyên gia về công nghệ thông tin đặc biệt quan tâm. Chính vì lý do đó mà *Cấu trúc dữ liệu và thuật toán* vẫn luôn là một chủ đề mà ngày càng được đầu tư nghiên cứu và phát triển, nó không bị lỗi thời trong bất cứ xu hướng nào của lĩnh vực Công nghệ thông tin, việc phát triển đề thuật toán ngày một hoàn thiện hơn gần như chưa bao giờ dừng lại và không có giới hạn.

Trước nhu cầu thiết thực cần tìm hiểu của bạn đọc, tác giả Trần Thông Quέ - giảng viên cao cấp đã có trên 35 năm kinh nghiệm giảng dạy ở lĩnh vực Công nghệ thông tin và luôn tâm huyết với chủ đề này đã phối hợp với Nhà xuất bản Thông tin và Truyền thông xuất bản bộ sách “*Cấu trúc dữ liệu và thuật toán - phân tích và cài đặt trên C/C++*”, gồm 2 tập.

Tập 1 gồm 5 chương, cụ thể như sau:

*Chương 1: Các khái niệm cơ bản về thuật toán và cấu trúc dữ liệu*

*Chương 2: Các thuật toán tìm kiếm trong và sắp xếp trong*

*Chương 3: Một số chiến lược thiết kế thuật toán*

*Chương 4: Các kiểu dữ liệu trừu tượng và biến nhớ động*

*Chương 5: Cấu trúc cây*

Với kinh nghiệm nhiều năm giảng dạy, tác giả đã lựa chọn cách tiếp cận đơn giản hóa các vấn đề phức tạp, từ cơ bản đến nâng cao.

Nội dung cuốn sách được trình bày một cách chi tiết, dễ hiểu, sau mỗi nội dung đều có ví dụ minh họa, sau mỗi thuật toán đều được đánh giá độ phức tạp và cài đặt demo. Cuốn sách còn bổ sung thêm một số thuật toán mà những cuốn sách đã xuất bản trước đây chưa có như: Thuật toán sắp xếp rung lắc, thuật toán sắp xếp với độ dài bước giảm dần... Chính vì vậy, cuốn sách này hướng tới đối tượng độc giả đông đảo, từ những bạn đọc mới làm quen tới những bạn đọc đã có kinh nghiệm nhiều năm trong lĩnh vực này. Đây thực sự là tài liệu bổ ích dành cho sinh viên, giảng viên, lập trình viên cũng như các chuyên gia về công nghệ thông tin có nhu cầu tìm hiểu và nghiên cứu.

Xin trân trọng giới thiệu cùng bạn đọc./.

**NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG**

## LỜI NÓI ĐẦU

Cấu trúc dữ liệu và Thuật toán (Data Structure and Algorithms) là môn học bắt buộc không những với mỗi sinh viên ngành Công nghệ Thông tin mà còn là môn học và môn thi quốc gia đầu vào bắt buộc với các nghiên cứu viên của ngành học đó. Nó là một trong các môn học khó của ngành CNTT, đặc biệt càng khó đối với hầu hết các sinh viên khi phải cài đặt một thuật toán hay một bài toán nào đó thuộc môn học này.

Dựa trên thực tế đó, bằng kinh nghiệm tích lũy được trong nhiều năm liên tục giảng dạy lý thuyết và thực hành môn học “*Data Structure and Algorithms*”, chúng tôi biên soạn cuốn sách này nhằm đáp ứng kiến thức một cách tối ưu cho mỗi sinh viên, các nghiên cứu viên ngành CNTT và cả các giảng viên phụ trách bộ môn này với thâm niên giảng dạy còn chưa nhiều.

Bất cứ sự thành công nào của một dự án Tin học đều là kết quả của việc kết hợp khéo léo giữa Cấu trúc dữ liệu và Thuật toán. Khẳng định này được chứng tỏ trong một công thức rất ngắn gọn mang tính triết lý đương đại nghề nghiệp:

**BIG DATA + BIG COMMUNITY = BIG RESULT**

**Dữ liệu lớn + Kết hợp lớn = Kết quả lớn**

Việc cài đặt các thuật toán cơ bản hoặc nổi tiếng hoặc khó được thực hiện trên các ngôn ngữ chuyên nghiệp đương đại là C/C++ nhằm giải đáp câu hỏi tồn tại trong đầu những người học là: Thực thi các thuật toán ấy trên máy tính điện tử như thế nào và sẽ cho kết quả ra sao?

Chúng tôi tin tưởng rằng tài liệu này sẽ rất bổ ích và hữu hiệu cho tất cả những ai có nhu cầu tìm hiểu về cấu trúc dữ liệu và giải thuật - một môn học tuy phức tạp song không kém phần lý thú, hấp dẫn.

Mặc dù đã có nhiều năm giảng dạy môn học này ở không ít trường đại học công và tư lập, song do những hạn chế chủ quan của bản thân nên cuốn sách sẽ khó tránh khỏi những khiếm khuyết, rất mong các độc giả xa gần cho các ý kiến “phản biện” để các lần tái bản sau sẽ ngày càng hoàn chỉnh hơn.

Xin chân thành cảm ơn các độc giả!

Tác giả  
**TRẦN THÔNG QUẾ**

# Chương 1

## CÁC KHÁI NIỆM CHUNG VỀ THUẬT TOÁN VÀ CẤU TRÚC DỮ LIỆU

### 1.1. MỘT VÀI KHÁI NIỆM VỀ THUẬT TOÁN

Về thuật ngữ, tiếng Anh “thuật toán” là “algorithm” có xuất sứ từ tên của nhà toán học cổ ở Trung Á là Abu Ja’fa Mohammed Ibn Musa Al Khowarizmi (khoảng năm 825). Trong khi nghiên cứu cách giải một loạt các bài toán đại số, Al Khowarizmi nhận thấy có một lớp những bài toán cùng kiểu có chung một cách giải. Và ông đã quy trình hóa cách giải lớp các bài toán đó bằng một tập hợp các bước giải có *trình tự trước sau* mà trước ông, các nhà toán học cổ đại chưa làm. Để kỷ niệm thành quả này của ông, các “hậu duệ” của ông đã lấy tên ông đặt cho những quy trình giải các bài toán cùng kiểu ấy là Al Khowarizmi. Dần dần theo thời gian, các nhà toán học đã *Anh hóa* thuật ngữ này thành **Algorithm** hiện đại, ngắn gọn và tiện dùng như ngày nay.

**1.1.1. Định nghĩa thuật toán:** Một cách đơn giản và dễ hiểu, song thiếu chặt chẽ, thiếu chính xác, khi hiểu thuật toán là cách giải bài toán. Về thuật ngữ này có khá nhiều tác giả đã phát biểu khác nhau, thể hiện các cách tiếp cận khác nhau về cùng một khái niệm. Trong các cách định nghĩa về thuật toán, định nghĩa sau đây ngắn gọn nhất, rõ ràng nhất và chặt chẽ nhất:

*“Thuật toán là một tập hợp hữu hạn các chỉ thị (Instructions) hoặc các thao tác (Operations) được sắp xếp theo một trật tự xác định để dạy cho con người, máy tính hoặc robot giải một bài toán hoặc giải quyết một vấn đề nào đó”.*

Từ định nghĩa này, ta trực tiếp suy ra các tính chất dưới đây của thuật toán:

#### 1.1.2. Các tính chất của thuật toán

a. **Tính hữu hạn (tính dừng):** Bất kỳ thuật toán nào cũng phải dừng sau *một số hữu hạn bước*. Tức là mọi thuật toán đều phải có **Bắt đầu** và **Kết thúc**. (Cuốn sách này không bàn về những thuật toán có vô hạn các chỉ thị (một ví dụ dễ thấy nhất là các thuật toán chạy lặp vòng (quẩn) vô kỳ hạn, tức nó không dừng). Có nhiều sinh viên mới học lập trình trên một ngôn ngữ bậc cao nào đó, khi thực hành rất hay mắc lỗi quên tính chất này. Cụ thể hơn: các sinh viên đó

quên viết từ khóa bắt đầu (trong C/C<sup>++</sup> là ký tự {) hoặc từ khóa kết thúc (trong C/C<sup>++</sup> là ký tự }) của ngôn ngữ đang thực hành cho chương trình đó, dẫn rằng trình dịch đã bắt lỗi mà cứ lóng ngóng mãi không sửa được lỗi cơ bản này. Hãy luôn thường trực trong đầu tính chất này của thuật toán khi lập trình.

**b. Tính xác định đơn nhất:** Tại mỗi bước của thuật toán, các chỉ thị phải rõ ràng, chuẩn xác. Các chỉ thị *không được diễn đạt bằng những ngôn từ mập mờ, hình như*, hoặc có thể hiểu theo những nghĩa khác nhau. Sự diễn đạt các chỉ thị *chỉ có một cách hiểu duy nhất*. Nói rõ hơn, trong cùng một điều kiện, **những người khác nhau, những máy tính khác nhau** cùng thực thi một chỉ thị nào đó của thuật toán thì phải **cho cùng một kết quả**.

**c. Tính đúng đắn:** Sau khi thực thi xong toàn bộ các bước của thuật toán, ta phải thu được kết quả mong muốn. Kết quả này phải được người xây dựng thuật toán xác định trước trong lúc xây dựng thuật toán.

**d. Tính phổ dụng:** Thuật toán phải giải được tất cả các bài toán thuộc cùng một lớp các bài toán cùng kiểu.

*Ví dụ 1.1:* Dân số Việt Nam hiện tại là 86 triệu người. Tỷ lệ bình quân tăng dân số hàng năm là 0,003%.

Tính xem sớm nhất đến năm nào dân số nước ta đạt ít nhất 100 triệu người?

*Ví dụ 1.2:* Tôi hiện có số vốn là 130 triệu đồng. Lãi suất hàng năm thu được là 8%. Tính xem sớm nhất đến năm nào tôi thu được số tiền (cả vốn lẫn lãi) là 300 triệu đồng?

Thuật toán xây dựng cho bài toán ở ví dụ 1.1 cũng phải dùng được cho bài toán ở ví dụ 1.2 và ngược lại (đó là hai bài toán cùng kiểu, dẫn rằng ý nghĩ của dữ liệu hoàn toàn khác nhau).

**e. Tính hiệu quả:** Tính chất này của thuật toán được đánh giá trên các chỉ tiêu sau:

- Thuật toán sử dụng tiết kiệm nhất các tài nguyên (resources) của máy tính như dung lượng bộ nhớ trong, bộ nhớ ngoài.

- Thực thi trên máy tính (chạy) nhanh nhất có thể được. (Chỉ tiêu này có tên khác là: thời gian chạy chương trình hoặc thời gian thực hiện thuật toán). Về mặt lý thuyết, chỉ tiêu này đáng chú ý nhất.

- Dễ cài đặt (Set Up) trên máy tính.

### 1.1.3. Ba đặc trưng của thuật toán

Bất kỳ thuật toán nào cũng có ba đặc trưng sau:

a. **Vào dữ liệu (Data Input)**. Hiển nhiên là bất kỳ thuật toán nào cũng phải có bộ dữ liệu vào. Đó là tập các dữ liệu phải nạp vào thuật toán khi nó bắt đầu được thực thi.

b. **Xử lý dữ liệu (Data Process)**. Mọi thao tác trên dữ liệu vào (nói riêng là tính toán) ta gọi là xử lý chúng để đạt được kết quả của bài toán. Công việc đặc trưng này về thực chất là nội dung chủ yếu (chính là các bước thực thi) của thuật toán. Không có đặc trưng này thì bất thành thuật toán.

c. **Xuất dữ liệu (Data Output)**. Cuối cùng thì thuật toán cũng phải cho ra kết quả giải bài toán hay giải quyết vấn đề. Không thể khác được! Việc này trong Tin học gọi là xuất dữ liệu. Các dữ liệu được xuất này có liên quan tới bộ dữ liệu vào và quá trình “chế biến” nó bởi các bước của thuật toán (công đoạn **Data Process**), đây là kết quả cần có sau khi chạy (thực thi) thuật toán.

### 1.1.4. Các cách diễn đạt thuật toán (Expressing Algorithms)

Dưới đây là những cách hay dùng để diễn đạt thuật toán.

a. **Diễn đạt thuật toán bằng ngôn ngữ tự nhiên**. Ngôn ngữ tự nhiên là ngôn ngữ mà con người của một quốc gia bất kỳ hàng ngày dùng để làm việc và giao tiếp. Ưu điểm của phương pháp này là dễ viết (vì ngôn ngữ tự nhiên quen thuộc từ nhỏ với mọi người). Nhược điểm độc nhất là máy tính “*không hiểu*” được thuật toán diễn đạt theo cách này. Do đó ta không thể chuyển giao thuật toán diễn đạt bằng ngôn ngữ tự nhiên cho máy tính.

*Ví dụ 1.3:* Diễn đạt bằng ngôn ngữ tự nhiên thuật toán tìm ước số chung lớn nhất (USCLN) của hai số nguyên dương tùy ý nhờ phương pháp trừ liên tiếp của Euclide.

Bước 1: (**Bắt đầu**) Nhập 2 số nguyên dương bất kỳ (vào thuật toán);

Bước 2: Chừng nào mà  $a-b \neq 0$  (tức  $a \neq b$ ) thì lặp các thao tác sau:

- Nếu  $a=b$  thì  $USCLN=a$ ; Đến bước 3;

- Nếu  $a \neq b$  thì:

- + Nếu  $a > b$  thì  $USCLN=a-b$ . Quay lại bước 2.

- + Ngược lại ( $a < b$ ) thì  $USCLN(a,b)=b-a$ . Quay lại bước 2;

Bước 3: (**Kết thúc**) Xuất kết quả.

**Có thể bạn chưa biết!** Trong máy tính bạn không cần phải có một đại lượng tên là USCLN (nó chỉ dùng khi giải bài toán trên giấy) mà bạn *dùng ngay biến a để lưu kết quả a-b (a=a-b)* và *biến b lưu kết quả b-a (b=b-a)* thay cho đại lượng USCLN. Trên máy tính ta viết lại thuật toán trên như sau:

Bước 1: (**Bắt đầu**) Nhập 2 số nguyên dương bất kỳ (vào thuật toán);

Bước 2: Chừng nào mà  $a-b \neq 0$  (tức  $a \neq b$ ) thì lặp các thao tác sau:

- Nếu  $a=b$  thì USCLN= $a$ ; Đến bước 3;

- Nếu  $a \neq b$  thì:

Nếu  $a>b$  thì  $a=a-b$ . Quay lại bước 2; { biến a lưu USCLN của (a,b) }

Còn thì  $b=b-a$ ;

Bước 3: (**Kết thúc**) Xuất kết quả.

**b. Diễn đạt thuật toán bằng mã giả (Pseudocode).** Mã giả là thứ ngôn ngữ do người thiết kế thuật toán “biết ra” (nói lịch sự hơn là “sáng tác”) để diễn đạt thuật toán. **Ưu điểm** nổi bật của phương pháp này là ngôn ngữ mã giả *không bị ràng buộc bởi những luật cú pháp nghiêm ngặt* của ngôn ngữ tự nhiên hoặc ngôn ngữ lập trình bậc cao. Vì vậy người viết mã giả rất *thoải mái*, nói cách khác là *bậc tự do của người viết rất lớn*. Những nhược điểm của phương pháp này là người viết phải có những lời giải thích các ký hiệu của riêng mình thì người đọc mới hiểu được, nhược điểm lớn nhất cũng giống như phương pháp diễn đạt nhờ ngôn ngữ tự nhiên là máy tính “*không hiểu*” được thuật toán.

**Ví dụ 1.4:** Dùng mã giả diễn đạt thuật toán cho bài toán ở ví dụ 1.3

Bước 1: (**Begin**) Input a, b ( $a, b \geq 0$ );

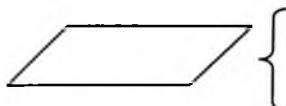
Bước 2: Loop  $a-b!=0$ ; (**Ký hiệu** “!=”: hiểu là khác hoặc không bằng, Loop là lặp lại)

- If( $a>b$ )  $a \leftarrow a-b$ ; (**Ký hiệu** “ $\leftarrow$ ”: hiểu là gán (giá trị của vé trái cho biến ở vé phải)

- Else (ngầm hiểu là  $b>a$ )  $b \leftarrow b-a$ ;

Bước 3: (**End**) Print Result.

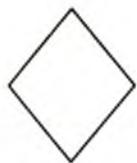
**c. Diễn đạt thuật toán bằng lưu đồ (Flowchart).** Cái gọi là lưu đồ để diễn đạt thuật toán là tập hợp các hình sau đây theo chuẩn ANSI (American National Standard Institute):



Để nhập dữ liệu vào thuật toán.



Để ghi công thức tính toán (trong ngôn ngữ lập trình bậc cao là các phép Gán).



Để ghi các điều kiện giải bài toán.



Để ghi bắt đầu hoặc kết thúc thuật toán.



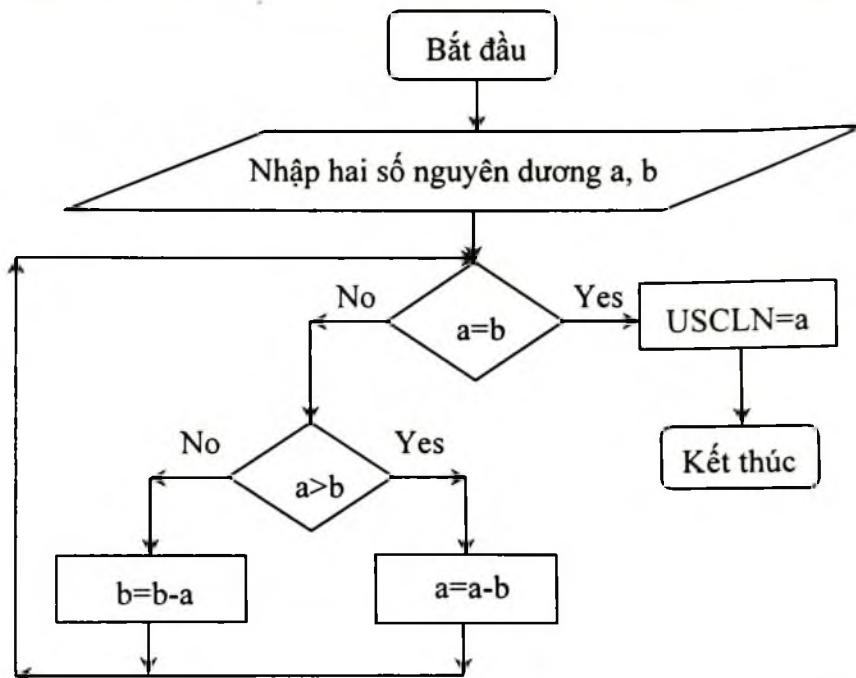
Cho biết mối liên hệ trước sau giữa các bước trong thuật toán và tiến trình thực thi thuật toán.

**Ưu điểm** đáng kể của phương pháp Flowchart là với những bài toán nhỏ, gọn thì nó cho ta thấy một cách trực quan thứ tự thực hiện các bước của thuật toán, mối liên hệ giữa các bước và ta *thấy ngay toàn cảnh thực thi bài toán* (hoặc công việc). Tuy nhiên với những bài toán lớn phức tạp có từ hàng trăm, ngàn... thao tác trở lên thì phương pháp này lại mất đi ưu điểm trên vì khi đó Flowchart trải ra (bị “phân mảnh”) trên nhiều trang giấy làm ta khó quan sát liên tục và tổng thể quy trình giải bài toán.

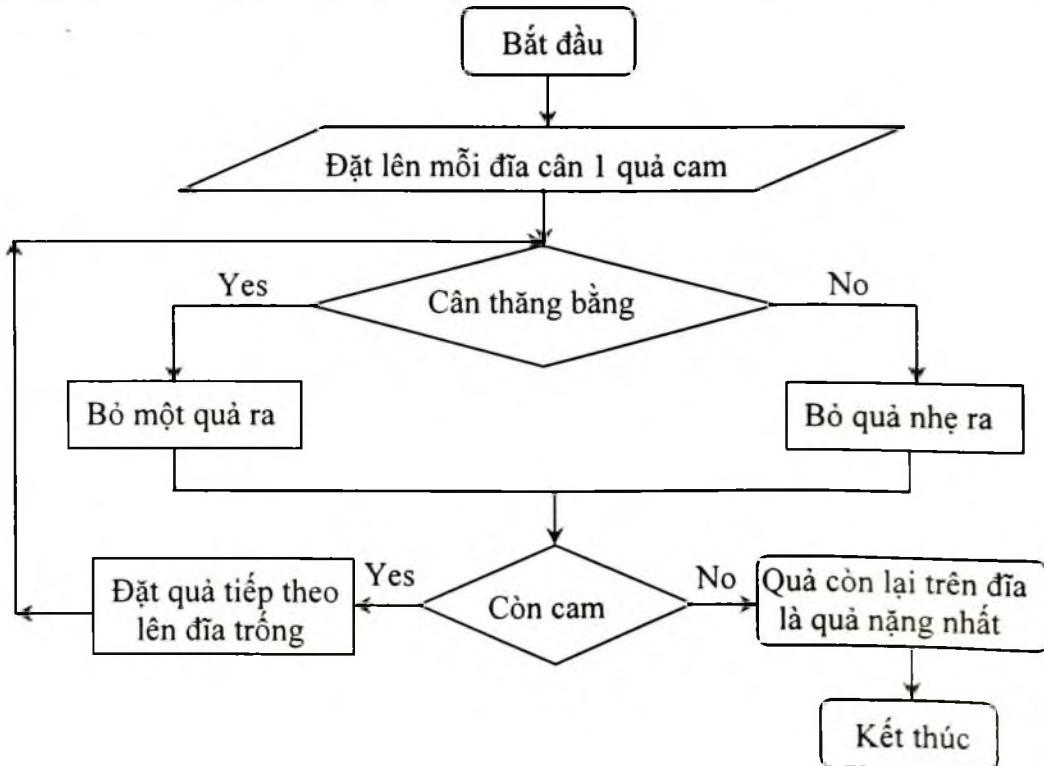
Còn một nhược điểm nữa của phương pháp này là xa lạ với nhiều người. Muốn dùng được nó hoặc hiểu nó khi đọc Flowchart thì phải được học trước. Nhược điểm cốt lõi của Flowchart là máy tính cũng **“không hiểu”** được thuật toán.

Tóm lại cả 3 phương pháp trên có một nhược điểm giống nhau là làm cho máy tính **“không hiểu”** được thuật toán. Ba phương pháp ấy chỉ dùng cho con người. Để khắc phục “sự cố” này, người ta dùng các ngôn ngữ lập trình bậc cao để diễn đạt thuật toán. Trước khi chuyển sang bàn về phương pháp thứ tư này ta xét ví dụ 1.5 diễn đạt bằng Flowchart.

Ví dụ 1.5: Dùng Flowchart diễn đạt thuật toán cho bài toán ở ví dụ 1.3



Ví dụ 1.6: Một rổ cam có N quả ( $N > 1$ ). Dùng cân có 2 đĩa cân xác định quả cam nặng nhất trong rổ. Hãy xây dựng Flowchart mô tả thuật toán của bài này.



#### d. Diễn đạt thuật toán bằng một trong các ngôn ngữ lập trình bậc cao

Như trên đã nói phương pháp này khắc phục được nhược điểm của ba phương pháp đã xét. Dùng ngôn ngữ lập trình bậc cao diễn đạt thuật toán khiến cho *cá người và máy tính đều hiểu được*. Nhờ vậy ta có thể *chuyển giao những bài toán khó, phức tạp cần giải trong một thời gian ngắn nhất có thể cho máy tính giải giúp*.

*Ví dụ 1.7:* Dùng ngôn ngữ lập trình bậc cao diễn đạt thuật toán của bài toán ở ví dụ 1.3 bằng ngôn ngữ C.

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a,b,tt;
    tt:clrscr();
    printf("Nhập a:"); scanf("%d", &a);
    printf("Nhập b:"); scanf("%d", &b);
    while (a>b)
    {
        if (a>b) a=a-b;
        else b=b-a;
    }
    printf(" Uscln=%d",a); printf("\n");
    printf("More (1/0)? . Go 1->tiếp, go 0->ngưng:");
    scanf("%d", &tt); if (tt==1) goto tt; }
```

Đoạn code diễn đạt  
thuật toán Euclide  
(dùng phép trừ liên  
tiếp)

#### 1.1.5. Đánh giá thời gian thực hiện (độ phức tạp) của thuật toán

Các chỉ tiêu khác thời gian thực hiện thuật toán (thời gian chạy chương trình) dùng để đánh giá hiệu quả của thuật toán phụ thuộc vào nhiều tham số cụ thể như: cấu hình máy tính, chương trình của một người cụ thể viết ra để thực hiện một thuật toán cụ thể (để cài đặt hay khó cài đặt trên máy tính) do vậy rất khó mà lý thuyết hóa nó được. Trong các chỉ tiêu đánh giá hiệu quả của thuật toán người ta thấy thời gian thực hiện thuật toán có thể lý thuyết hóa nó để xem xét hiệu quả của thuật toán. Thời gian thực hiện thuật toán phụ thuộc và các yếu tố sau:

- Kích cỡ (thường là số lượng) của bộ dữ liệu vào.
- Tốc độ xử lý của CPU.
- Chương trình dịch (để dịch chương trình nguồn (viết bằng ngôn ngữ lập trình bậc cao) thành mã máy).

Để đơn giản do đó tiện dùng, người ta cũng không thể dùng thời gian thực hiện thuật toán như một hàm phụ thuộc ba tham số nêu trên. Với mục đích ấy, các nhà lý thuyết thuật toán *đã xem thời gian thực hiện thuật toán như một hàm chỉ phụ thuộc một tham số duy nhất đó là kích cỡ dữ liệu vào* (để gọn sau đây ta chỉ gọi là *cỡ dữ liệu*).

Cỡ dữ liệu vào phụ thuộc vào kiểu dữ liệu mà ta chọn dùng cho thuật toán. Trong bài toán ở ví dụ 1, thì cỡ dữ liệu là hai (2) số nguyên dương. Ví dụ ta có bài toán tính tổng của n số tự nhiên thì cỡ dữ liệu là n. Phổ biến nhất, cỡ của bộ dữ liệu là số nguyên dương n. Ta ký hiệu  $T(n)$  là hàm để đánh giá thời gian thực hiện thuật toán (một cái tên khác cũng rất hay dùng là *độ phức tạp của thuật toán, ta sẽ viết tắt là ĐPT*). Bước đầu để bạn đọc dễ hiểu hơn khái niệm ĐPT ta xét ví dụ cần tìm một số nguyên trong một danh sách có n số nguyên (khi học các môn lập trình cơ sở bạn đã quá quen với bài toán này). Nếu ta thực hiện phép tìm kiếm tuần tự-phép tìm kiếm đơn giản nhất - thì ta phải làm n phép so sánh (trong tình huống xấu nhất: ta phải duyệt toàn bộ dãy số; tình huống tốt nhất là số cần tìm nằm ngay ở đầu danh sách, khi đó ta chỉ cần 1 phép so sánh). Ở bài toán này, nếu ta coi ĐPT là số phép so sánh thì  $\text{ĐPT}(n)=n$  (*tình huống tồi tệ nhất*) và  $\text{ĐPT}(n)=1$  (*tình huống tốt nhất*). Như vậy sẽ phải nghĩ đến *ĐPT trung bình*. Đây lại là giá trị *khó tính toán nhất* của ĐPT *trong đa số các bài toán thực tế*.

Về mặt toán học, ta có thể xem ĐPT là số các toán tử cơ bản cần làm khi thực thi thuật toán. Ví dụ các toán tử số học +, -, \*, / (chia lấy giá thực), % (chia lấy phần dư); các toán tử quan hệ:

$<, >, ==, \geq, \leq$  của ngôn ngữ C/C<sup>++</sup>.

#### 1.1.6. Khái niệm ô lớn (Big\_O, ký hiệu là O())

ĐPT của thuật toán được đánh giá khá thuận tiện khi sử dụng khái niệm Big\_O, lần đầu tiên được nhà toán học Đức Edmund Landau đưa ra vào năm 1892. Với khái niệm này khi đánh giá ĐPT người ta chỉ quan tâm đến cỡ của dữ liệu vào mà không quan tâm đến tốc độ của CPU, năng lực của chương trình dịch (Translater). Dưới đây là định nghĩa khái niệm Big\_O.

**a. Định nghĩa Big\_O.** Cho hai hàm  $f(x)$  và  $g(x)$  với  $x$  thuộc tập các số nguyên hoặc các số thực. Ta gọi  $f(x)$  là  $O(g(x))$  (đọc là ô lớn của  $g(x)$ ) nếu chọn được hai hằng số  $C$  và  $k$  sao cho:

$$\left| f(x) \right| \leq C \cdot |g(x)| \quad (\text{hệ thức Landau})$$

$$\forall x > k$$

Có thể cặp  $C, k$  không là duy nhất, ý nghĩa của hệ thức này là: Để xác định  $f(x)$  là ô lớn của  $g(x)$  ta chỉ cần tìm cặp hằng số  $C, k$  sao cho thỏa hệ thức Landau với  $\forall x > k$ !

**Ví dụ 1.8:** Chứng tỏ rằng  $f(x)=x^2+2x+1$  có ô lớn  $O(g(x))=O(x^2)$ .

**Giải.** Hiển nhiên với  $\forall x > 2$  ( $2$  là  $k$ -theo định nghĩa của Big\_O) ta có:

$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2 \text{ ở đây } C=3 \text{ và như dòng trên } k=2.$$

Vậy  $x^2+2x+1$  có  $O(g(x))=O(x^2)$ . Người ta cũng nói  $O(x^2)$  là ĐPT (thời gian thực hiện) của thuật toán tính hàm  $f(x)=x^2+2x+1$ .

**Ví dụ 1.9:** Chứng tỏ rằng ĐPT của  $f(x)=11x^2$  là  $O(x^3)$ .

**Giải.** Thật vậy với mọi  $x > 11$  ( $11$  là  $k$ ) ta có  $11x^2 < x^3$ . Vậy theo hệ thức Landau ta được  $O(x^3)$  là ĐPT của  $11x^2$  (ở ví dụ này  $C=1, k=11$ ).

### b. Một số bài tập tổng quát

**Bài tập 1.** Cho  $f(x)$  là một đa thức bậc  $n$ , với các hệ  $a_i$  ( $i=0, 1, 2, \dots, n$ ) là các số thực. Chứng minh rằng ĐPT của  $f(x)$  là  $O(x^n)$ .

**Giải.** Ta dùng bất đẳng thức tam giác. Hiển nhiên với  $\forall x > 1$  có:

$$\begin{aligned} |f(x)| &= |a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0| \\ &\leq |a_n| |x^n| + |a_{n-1}| |x^{n-1}| + \dots + |a_1| |x| + |a_0| \\ &= x^n \left( |a_n| + \frac{|a_{n-1}|}{x} + \dots + \frac{|a_1|}{x^{n-1}} + \frac{|a_0|}{x^n} \right) \\ &\leq x^n \underbrace{\left( |a_n| + |a_{n-1}| + \dots + |a_1| + |a_0| \right)}_{\text{Chọn tổng này làm hằng số } C} \end{aligned}$$

Chọn tổng này làm hằng số  $C$

Điều này chứng tỏ  $|f(x)| \leq C \cdot x^n$  với  $\forall x > 1$  ( $k=1$ ), nghĩa là ĐPT của đa thức bậc  $n$  với các hệ số thực là  $O(x^n)$ .

Từ bài tập 2 ta xét một số ví dụ về ĐPT của các hàm có miền xác định là các số nguyên dương.

**Bài tập 2.** Xác định ĐPT của tổng n số tự nhiên đầu tiên:

$$S(n) = 1+2+3+\dots+n$$

*Giải.* Rõ ràng mỗi số nguyên dương trong tổng không vượt quá n, nên:

$$S(n) \leq n+n+\dots+n = n^2 \quad (C=1 \text{ và } k=1)$$

Vậy ĐPT của tổng S(n) là O( $n^2$ ) với C=1 và k=1.

**Bài tập 3.** Đánh giá Big\_O của phép tính giai thừa  $n!=1.2.3.\dots.n$ .

*Giải.* Vì mỗi số trong tích trên không vượt quá n nên:

$n!=1.2.3\dots.n \leq \underbrace{n.n.\dots.n}_{\text{có } n \text{ số } n} = n^n = C.n^n \quad (*) \quad (C=1, k=1)$  suy ra ĐPT của n! là O( $n^n$ )

**Bài tập 4.** Đánh giá ĐPT của phép logarithm của phép tính giai thừa. Nghĩa là tính O(logn!).

*Giải.* Lấy logarithm 2 vế hệ thức (\*) ở trên ta được:

$$\log n! \leq \log n^n = n \log n \text{ suy ra ĐPT của } \log n! \text{ là } O(n \log n).$$

### 1.1.7. Hai luật cơ bản của Big\_O

a. **Độ phức tạp của hàm tổng.** Độ phức tạp của hàm tổng cho bởi định lý sau:

**Định lý 1:** Hàm  $f_1(x)$  có ĐPT là  $O(g_1(x))$ , hàm  $f_2(x)$  có ĐPT là  $O(g_2(x))$ . Khi đó hàm tổng  $(f_1+f_2)(x)$  có ĐPT là  $O(\max(g_1(x), g_2(x)))$ .

**Chứng minh:** Theo định nghĩa của Big\_O, ta có thể xác định được các hằng số  $C_1, C_2, k_1, k_2$  sao cho:

$$|f_1(x)| \leq C_1 |g_1(x)| \text{ với } \forall x > k_1$$

$$\text{Và } |f_2(x)| \leq C_2 |g_2(x)| \text{ với } \forall x > k_2$$

Áp dụng bất đẳng thức tam giác, ta có:

$$|(f_1+f_2)(x)| = |f_1(x)+f_2(x)| \leq |f_1(x)| + |f_2(x)|$$

Khi chọn x lớn hơn cả  $k_1$  và  $k_2$  ta được:

$$\begin{aligned} |f_1(x)| + |f_2(x)| &\leq C_1 |g_1(x)| + C_2 |g_2(x)| \leq C_1 |g(x)| + C_2 |g(x)| \\ &= (C_1 + C_2)g(x) = Cg(x) \end{aligned}$$

Trong đó:  $C = C_1 + C_2$  và  $g(x) = \max(g_1(x), g_2(x))$ . Tóm lại:

$|(f_1+f_2)(x)| \leq C |g(x)|$  với  $\forall x > k$  mà  $k = \max(k_1, k_2)$ . Định lý được chứng minh.

**b. Độ phức tạp của hàm tích.** Định lý dưới đây cho phép ta xác định được ĐPT của hàm tích.

**Định lý 2:** Hàm  $f_1(x)$  có ĐPT là  $O(g_1(x))$ , hàm  $f_2(x)$  có ĐPT là  $O(g_2(x))$ . Khi đó hàm tích  $(f_1 \cdot f_2)(x)$  có ĐPT là  $O(g_1(x) \cdot g_2(x))$ .

**Chứng minh:** Khi  $x > k = \max(k_1, k_2)$  ta có (theo định nghĩa của Big\_O):

$$\begin{aligned} |(f_1 \cdot f_2)(x)| &= |f_1(x)| \cdot |f_2(x)| \leq C_1 |g_1(x)| C_2 |g_2(x)| \leq C_1 C_2 |(g_1 \cdot g_2)(x)| \\ &= C |(g_1 \cdot g_2)(x)| \text{ ở đây } C = C_1 \cdot C_2. \end{aligned}$$

Định lý được chứng minh.

### c. Các trường hợp của $O(n)$

- Nếu  $O(n)=1$  thì có nghĩa là ĐPT của thuật toán bị chặn trên bởi một hằng số, khi đó ta nói *ĐPT của thuật toán là hằng số*.

- Nếu  $O(n)=n$  thì tức là từ một  $n_0$  nào đó trở đi ĐPT của thuật toán tỷ lệ với cỡ của dữ liệu vào và ta gọi là *ĐPT là tuyến tính*.

Dưới đây là bảng kê các ĐPT thông thường nhất của thuật toán và xếp theo trật tự tăng dần của nó:

Độ phức tạp thuật toán	Tên gọi
$O(1)$	Hằng số
$O(\log n)$	Logarithm
$O(n)$	Tuyến tính
$O(n \log n)$	Tích của n với logn
$O(n^2)$	Bình phương
$O(n^3)$	Lập phương
$O(2^n)$	Mũ n cơ số 2
$O(n^n)$	Mũ n cơ số n

### d. Bài luyện tập

**Bài tập 1.** Chứng minh rằng ĐPT của  $\frac{x^2 + 1}{x + 1}$  là  $O(x)$ .

Dưới đây là cách giải không “chân phương”:

$$\frac{x^2 + 1}{x + 1} = x - 1 + \frac{2}{x + 1} < x \text{ với } \forall x > 1 \text{ do đó ĐPT của phân thức đã}$$

cho là  $O(x)$ .

**Bài tập 2.** Xác định ĐPT của hàm sau  $f(x) = 3n \log(n!) + (n^2 + 3)\log n$  với  $n$  nguyên dương.

Theo kết quả bài tập 4 ở mục 1.1.6.b ta có ĐPT của  $\log(n!)$  là  $O(n\log n)$ .  
 Hiện nhiên ĐPT của  $3n$  là  $O(n)$ . Vậy theo định lý 2 trên đây ta được ĐPT của  $3n\log(n!)$  là  $O(n^2\log n)$ . Ta xác định nốt ĐPT của  $(n^2+3)\log n$ . Vì rõ ràng  $n^2 + 3 < 2n^2$  khi  $n > 2$  (ở đây  $k=2$ )  $\rightarrow$  ĐPT của  $n^2 + 3$  là  $O(n^2)$ .

Từ định lý 2 có ĐPT của  $(n^2+3)\log n$  là  $O(n^2\log n)$ . Cuối cùng từ định lý 1 trên đây ta có ĐPT của hàm đã cho là  $O(n^2\log n)$ .

### 1.1.8. Đánh giá ĐPT của các câu lệnh cơ bản của C/C<sup>++</sup>

**a. Mở đầu.** Để đánh giá ĐPT của một chương trình đơn giản (không có lời gọi hàm) viết trên C/C<sup>++</sup>, trước hết ta phải biết cách đánh giá ĐPT của các câu lệnh cơ bản trên C/C<sup>++</sup>. Muốn thế ta nhắc lại cú pháp (syntax) của các câu lệnh đó trong C/C<sup>++</sup>.

### b. Các câu lệnh cơ bản của C/C<sup>++</sup>

#### **Các câu lệnh chuyển điều khiển (các câu lệnh rẽ nhánh có điều kiện)**

- Lệnh if: if (Biểu thức Logic) <Lệnh>;

(Để ngắn gọn, sau này ta ký hiệu btlg cho “Biểu\_thức\_Logic” và S cho “Lệnh”).

- Lệnh if ... else: if (btlg) <S1>; else <S2>;

(trong C/C++, trước else có chấm phẩy (;), còn trong Pascal điều này cấm kỵ)

- Lệnh switch . . . case: switch (bt1g)

```
[default <S>; break;
}
```

(ở mục này không ôn lại các lệnh rẽ nhánh vô điều kiện)

### Các câu lệnh lặp

- Lệnh for: for (Biểu\_thúc\_1; Biểu\_thúc\_2; Biểu\_thúc\_3)  
{<S>; . .}

- Lệnh while: while (Biểu\_thúc)
 

```
{
      <S>;
}
```

- Lệnh do...while

```
do
{
  <S>;
}
while (Biểu_thúc);
```

### c. Độ phức tạp của các lệnh đơn hoặc các câu lệnh cơ bản của C

Để đánh giá ĐPT của một chương trình ta dùng phương pháp đệ quy dựa trên ĐPT của các lệnh cơ bản nói trên và giả sử rằng **lệnh gán không chứa lời gọi hàm**:

- Thời gian thực hiện (ĐPT) của các lệnh đơn: gán, đọc, ghi, return(một trị số) là O(1).
- ĐPT của lệnh hợp thành xác định bởi luật tổng (của ĐPT ).
- Lệnh if: Nếu ĐPT của các lệnh S<sub>1</sub> là O(g<sub>1</sub>(n)) và của S<sub>2</sub> là O(g<sub>2</sub>(n)) thì ĐPT của lệnh if là O(Max(g<sub>1</sub>(n), g<sub>2</sub>(n))).
- Lệnh switch...case: Có ĐPT được đánh giá tương tự lệnh if.
- Lệnh while: Giả sử ĐPT khi thực hiện lệnh S trong thân vòng lặp while là O(g(n)) còn số tối đa các lần thực hiện lệnh S khi kết thúc lặp while là k thì ĐPT của lặp while là O(k\*g(n));
- Lệnh do...while và lệnh for: Đánh giá ĐPT tương tự như lệnh while.

#### d. ĐPT của lệnh gán có lời gọi hàm (tức là ĐPT của các hàm đệ quy)

- Trước hết cần nói rằng việc đánh giá ĐPT của các hàm phi đệ quy chỉ cần áp dụng các quy tắc ở mục 1.8.c trên đây.

- Còn việc đánh giá ĐPT các hàm đệ quy (trong C/C<sup>++</sup> chỉ có hàm, *không có thủ tục!!!*) thì phức tạp hơn nhiều. Để làm việc này ta xuất phát từ một ví dụ cụ thể đơn giản sau, rồi mở rộng thành tổng quát: hàm đệ quy tính giai thừa của số nguyên không âm n.

```
long int gtdq(int n)
{ if (n==0 || n==1) return1;
  else return (n*gtdq(n-1));
}
```

Trong hàm này, cỡ của dữ liệu vào là n. Giả sử ĐPT của hàm ấy là g(n). Với n=1 chỉ cần gán gtdq=1 do vậy g(1)=1. Với n>1 hiển nhiên ta thực hiện lệnh gán gtdq=n\*gtdq(n-1), khi đó ĐPT của g(n) là: g(n)=O(1)+g(n-1). Vì O(1) là ĐPT hằng số nên để gọn ta thay các O(1) bằng các hằng C<sub>1</sub>, C<sub>2</sub>... nào đó:

$$\begin{aligned} g(1) &= C_1; \\ g(n) &= C_2 + g(n-1); \end{aligned}$$

Để tìm g(n) từ phương trình đệ quy trên đây ta dùng cách thế lặp. Ta có phương trình đệ quy:

$$g(m)=C_2+g(m-1) \text{ với } m>1$$

Thay m lần lượt bởi các trị số 2,3,4,...,n-1 ta có hệ phương trình sau:

$$\begin{aligned} g(2) &= C_2 + g(1) \\ g(3) &= C_2 + g(2) \\ &\dots\dots\dots \\ g(n-1) &= C_2 + g(n-2) \\ g(n) &= C_2 + g(n-1) \end{aligned}$$

Dùng phép thế lặp đã nói trên ta được: g(n)=(n-1)C<sub>2</sub>+g(1) hay g(n)=(n-1)C<sub>2</sub>+C<sub>1</sub> trong đó C<sub>1</sub>, C<sub>2</sub> là các hằng nào đó đã chọn trước. Đến đây rõ ràng ĐPT của g(n) trong ví dụ của ta là O(n).

*Tổng quát:* Đệ quy có nhiều loại, để đơn giản, ở đây ta chỉ xét đệ quy trực tiếp (nghĩa là loại đệ quy mà hàm chỉ gọi tới chính nó, không qua một hàm khác). Giả sử ĐPT của hàm là g(n), n-cỡ của dữ liệu vào, và ĐPT của các lời

gọi đệ quy hàm là  $g(m)$  với  $m < n$ . Ta sẽ đánh giá ĐPT  $g(n_0)$  với  $n_0$  là cỡ dữ liệu vào nhỏ nhất có thể được (ở ví dụ trên là  $g(1)$ ), sau đó đánh giá ĐPT của thân hàm theo các quy tắc nêu ở mục 1.1.8.c ở trên ta được phương trình đệ quy sau:

$$g(n) = f(g(m_1), g(m_2), g(m_k)) \text{ với } m_1, m_2, m_3, \dots, m_k < n$$

Giải phương trình đệ quy trên ta được ĐPT  $g(n)$ . Trong nhiều trường hợp việc giải phương trình đệ quy trên là rất phức tạp.

### 1.1.9. Đánh giá ĐPT của một số chương trình (thuật toán) cơ bản

a. *ĐPT của chương trình thực thi thuật toán tìm ước số chung lớn nhất* của hai số nguyên dương bằng thuật toán Euclide dùng phương pháp *chia liên tiếp*.

```
int uscln(int x, int y)
{
    int du;
    while (y != 0)
    {
        du=x%y;           //1)
        x=y;              //2)
        y=du;             //3)
    }
    return (x); //4)
}
```

Theo các nguyên tắc 1.1.8.c, các lệnh đơn 1) → 4) có ĐPT là  $O(1)=\text{hằng}$ . Ta còn phải đánh giá số lần nhiều nhất thực hiện lặp khối lệnh trong thân while. Số lần đó phụ thuộc vào số nhỏ nhất trong 2 số  $x, y$ . Không ảnh hưởng gì đến tính tổng quát của vấn đề, ta có thể giả sử  $x \geq y > 0$ , do đó cỡ của dữ liệu vào là  $y$ . Ta luôn có thể biểu diễn:

$$x = y \cdot q_1 + r_1 \text{ với } 0 \leq r_1 < y$$

$$y = r_1 q_2 + r_2 \text{ với } 0 \leq r_2 < r_1$$

Nếu  $r_1 \leq y/2$  thì  $r_2 \leq r_1 \leq y/2$  nghĩa là  $r_2 \leq y/2$ . Nếu  $r_1 \geq y/2$  thì  $q_2=1$ , tức là  $y = r_1 + r_2$  suy ra  $r_2 \leq y/2$ . Tóm lại *bao giờ ta cũng có  $r_2 \leq y/2$* . Như vậy cứ hai lần thực hiện lặp khối lệnh thì số dư giảm đi một nửa của  $y$ . Ký hiệu  $k$  là số nguyên dương lớn nhất sao cho  $2^k \leq y$ . Nhờ một vài biến đổi sơ cấp ta có:  $2k+1 \leq 2\log_2 y + 1$ . Nghĩa là thời gian thực hiện khối lệnh là  $O(\log_2 y)$ . Đó cũng chính là ĐPT của thuật toán Euclide dùng phương pháp chia liên tiếp.

### b. ĐPT của thuật toán đệ quy xác định dãy Fibonacci với độ dài n cho trước

Trước hết ta nhắc lại khái niệm dãy Fibonacci: Dãy Fibonacci là dãy các số nguyên dương có cấu trúc sau:

1 1 2 3 5 8 13 21 34 55 89 144 ...

Hai số hạng đầu tiên của dãy này luôn luôn là 1. Từ số hạng thứ hai trở đi, các số hạng luôn bằng tổng hai số đứng ngay trước nó. Nói rõ hơn, gọi  $s_i$  là số hạng hiện thời ( $i \leq n$ ,  $n$  là độ dài cho trước của dãy này) thì hiển nhiên:

$$s_i = s_{i-1} + s_{i-2} \text{ với } i \geq 2$$

Từ đó suy ra cách thiết kế hàm đệ quy xác định dãy Fibonacci với độ dài n cho trước:

```
int fib(int i)
{
    if (i==1 || i==2) return(1);
    else return f(i-1)+f(i-2);
}
```

Dưới đây ta sẽ đánh giá ĐPT của hàm đệ quy fib; gọi  $T(n)$  là thời gian thực hiện hàm này, ta có các hệ thức sau:

$$T(i)=c \text{ với } i<2 \text{ (c là hằng số)}$$

$$T(i)=T(i-1)+T(i-2) \text{ với } i \geq 2$$

Việc giải hệ phương trình đệ quy trên đây khá phức tạp. Ở đây ta không nêu cách giải này, chỉ cốt tập trung vào việc đánh giá ĐPT của hàm fib trên đây. Nhờ cách giải phương trình đặc trưng của hệ này người ta đã tìm được  $T(i)$  của Fib là  $O(\alpha^i)$  với  $\alpha=(1+\sqrt{5})/2$ . Vậy  $T(i)$  của hàm đệ quy Fib trên đây tỷ lệ thuận với lũy thừa mũ của cỡ dữ liệu. Đó là ĐPT tăng trưởng rất nhanh và cực lớn khi cỡ dữ liệu cực lớn, nên nó hầu như không dùng với cỡ dữ liệu lớn.

Dưới đây ta xét một thuật toán khác cũng xác định dãy Fibonacci với độ dài cho trước n, với ĐPT đơn giản hơn (bản code này là hoàn chỉnh - các bạn có thể test).

```
#include <stdio.h>
#include <conio.h>
```

```

int fibo(int n) { /* Đây là hàm mô tả thuật toán phi đệ quy tính số
int i;          }   hạng thứ n của dãy Fibonacci và không dùng
int a=1;    //1)   mảng chỉ dùng các biến đơn*/
int b=0;    //2)
int c;
for (i=1; i<=n; i++) //3)
{ c=b+a;           //4)
  a=b;             //5)
  b=c;             //6)
}
return c;           //7)
}

main()
{ int i,n; clrscr();
  printf("\n Cho do dai day=");
  scanf("%d",&n);
  printf("\nDay Fibonacci can tim ung voi do dai
         vua nhap=%d la:\n",n);
  for(i=1; i<=n; i++)
    printf(" %3d ",fibo(i));
  getch();
}

```

Theo các quy tắc 1.1.8.c, các lệnh từ 1) đến 7) đều có ĐPT là  $O(1)$ = hằng; còn lệnh lặp for thực hiện lặp n lần ( $n$ -cỡ của dữ liệu vào), vậy thời gian thực hiện lệnh for là  $O(n)$ - đó cũng là ĐPT của lệnh for này.

Kết hợp lại, hàm fibo *phi đệ quy* có ĐPT là  $O(n) \ll O(\alpha^n)$  của thuật toán *đệ quy* fib.

## 1.2. KIẾU DỮ LIỆU VÀ CẤU TRÚC DỮ LIỆU

### 1.2.1. Kiểu dữ liệu là gì?

Kiểu dữ liệu là một *sự quy định về miền tồn tại giá trị của kiểu dữ liệu ấy và tập các phép (operations) tác động lên miền giá trị đó*. Ta có thể “toán học”

hóa định nghĩa trên đây như sau: Kiểu dữ liệu T là tập gồm hai tập con ký hiệu  $T=\{V, O\}$  (T viết tắt của Type, V viết tắt của Value; O viết tắt của Operations); trong đó:

- V: Tập (miền) các giá trị của kiểu T (*do một ngôn ngữ lập trình bậc cao quy định*).

- O: Tập các phép tác động lên miền giá trị V của kiểu T

*Ví dụ 1.10:* Trong ngôn ngữ lập trình C/C<sup>++</sup> kiểu dữ liệu long có  $V=\{-2147483648, 2147483677\}$  với tập  $O=\{+, -, *, /, \%\}$ ; còn kiểu bit có  $V=\{0, 1\}$  và  $O=\{\sim, \&, |, ^, <<, >>\}$

### 1.2.2. Định nghĩa cấu trúc dữ liệu

Cấu trúc dữ liệu là tập hợp các phần tử mà mỗi phần tử là một kiểu dữ liệu cơ sở hoặc lại là một cấu trúc dữ liệu khác của một ngôn ngữ lập trình bậc cao nào đó hoặc của người dùng. Các phần tử trong cấu trúc dữ liệu được tổ chức theo một phương thức xác định.

*Ví dụ 1.11:* Trong C/C<sup>++</sup>, cấu trúc dữ liệu mảng gồm các phần tử được tổ chức theo một trật tự các chỉ số tăng dần từ trái qua phải, bắt đầu từ số 0. Mỗi phần tử chiếm một ô nhớ (Cell Memory) trong Ram. Các phần tử của cấu trúc này có thể là kiểu dữ liệu cơ sở của C/C<sup>++</sup> như: Kiểu char, kiểu long int , kiểu logic..., hoặc cũng có thể là một cấu trúc dữ liệu khác như các *bản ghi* (trong C/C<sup>++</sup> là *struct*), khi đó ta có mảng các bản ghi.

*Ví dụ 1.12:* Cấu trúc dữ liệu *tệp tin* của C/C<sup>++</sup> lại có hai phương thức tổ chức: tổ chức theo cách truy cập (access) *tuần tự* hoặc tổ chức theo cách truy cập *ngẫu nhiên*.

### 1.2.3. Các kiểu dữ liệu và cấu trúc dữ liệu trong C/C<sup>++</sup>

#### a. Các kiểu dữ liệu cơ sở

*Lưu ý:*

- Kiểu char trong C/C<sup>++</sup> có thể dùng như ký tự hoặc như số nguyên 1 byte tùy ngữ cảnh cụ thể (đó là điều khác với Pascal và Foxpro...).

- Trong C/C<sup>++</sup> không có kiểu Logic (Boolean), song người dùng có thể tự định nghĩa kiểu này nhờ toán tử tiền xử lý #define để đồng nhất 1 với True và 0 với False như sau: #define True 1; #define False 0.

Tên kiểu	Miền giá trị	Số bytes chiếm dụng bộ nhớ
Unsigned char	0 → 255	1 Byte
char	-128 → 127	1 Byte
unsigned int	0 → 65535	2 Byte
int	-32768 → 32767	2 Byte
unsigned long	0 → $2^{32}-1$	4 Byte
long	$-2^{32} \rightarrow 2^{31}-1$	4 Byte
float	3.4E-38 → 3.4E38	4 Byte
double	1.7E-308 → 1.7E308	8 Byte
long double	3.4E-4932 → 1.1E4932	10 Byte

### b. Các kiểu dữ liệu cơ sở có cấu trúc

**Mảng (Array).** Mảng là kiểu dữ liệu có cấu trúc gồm **số hữu hạn** các phần tử **cùng kiểu** (cơ sở). Số phần tử của mảng được ấn định ngay từ khi khai báo mảng. Việc truy cập vào mỗi phần tử của mảng là trực tiếp và thông qua chỉ số (số thứ tự-Index-) của nó ghi giữa 2 ngoặc vuông [ ].

Có các loại mảng: mảng 1 chiều, mảng 2 chiều,... mảng n chiều. Trong đó đại đa số các bài toán thực tế chỉ dùng mảng 1 chiều và mảng 2 chiều.

*Cú pháp khai báo mảng một chiều*

<Kiểu dữ liệu> <Tên biến mảng>[<số phần tử>]

hoặc là: <Kiểu dữ liệu> <Tên biến mảng>[ ]=(phần\_tử\_1, phần\_tử\_2,...)

Để dễ hiểu, dễ nhớ bạn cần biết hình ảnh hình học của mảng 1 chiều có cấu trúc như sau (giống cuộn phim đã chụp hình):

Chỉ số của các phần tử      i=0      1      2 .....      n-2      n-1



Hình 1.1. Hình ảnh hình học (hay cấu trúc) của mảng 1 chiều

Trong đó n là số phần tử của mảng, còn gọi là **kích thước của mảng**.

Khi bạn khai báo xong một biến mảng 1 chiều, lập tức hệ điều hành cấp phát một loạt ô nhớ (cells) trong Ram với cấu trúc trên. Mỗi ô nhớ chứa 1 phần

tử duy nhất của mảng. Bạn cũng có thể quan niệm mảng 1 chiều là ma trận có duy nhất 1 hàng và n cột.

*Ví dụ 1.13:* Dùng mảng một chiều lưu trữ điểm trung bình học kỳ 1 của 60 sinh viên:

```
float Dtb_ky_1[60];
```

- Dùng mảng một chiều lưu điểm thi môn Lập trình với C của 6 sinh viên thi không đạt:

```
float Diem_Lt_C[]={3.5, 1, 2, 0, 4.5, 4})
```

Việc truy cập vào mỗi phần tử là truy cập trực tiếp theo chỉ số của phần tử đó.

Chẳng hạn, để xem giá trị của phần tử thứ 3 trong mảng này ta viết:

```
cout<<Diem_lt_C[3]
```

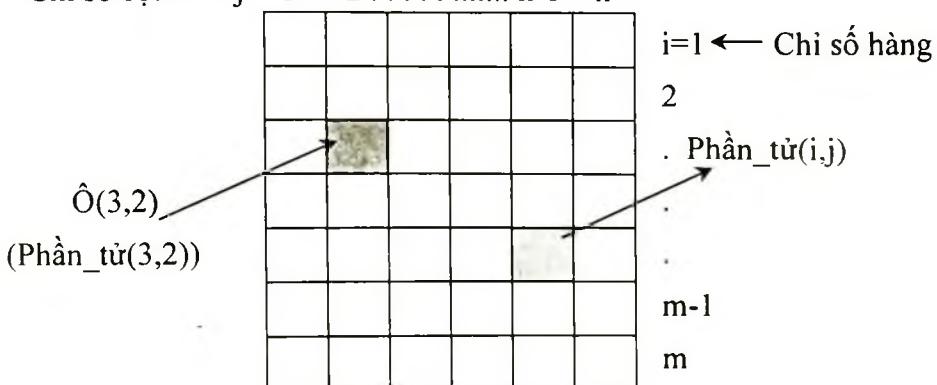
## *Cú pháp khai báo mang hai chiều*

**<Kiểu dữ liệu> <Tên biến mảng> [<số\_phần\_tử\_của\_hàng>] [<số\_phần\_tử\_của\_cột>];  
hoặc là:**

<Kiểu dữ liệu> <Tên biến mảng>[ ] [ ]={{pt11,pt12,...}, {pt21, pt22, ...}};

Để dễ hiểu, dễ nhớ bạn cần biết hình ảnh hình học của mảng 2 chiều là một lưới ô vuông có  $m$  hàng và  $n$  cột ( $m, n$ : hữu hạn). Mỗi ô vuông chứa một phần tử duy nhất của mảng. Chỉ số hàng và chỉ số cột của ô ấy là tọa độ (địa chỉ) của ô này.

Chỉ số cột  $\rightarrow j = 1 \quad 2 \dots \dots \dots n-1 \quad n$



Hình 1.2 Hình ảnh hình học (hay cấu trúc) của mảng 2 chiều

*Ví dụ 1.14:* float a[20][30] → khai báo mảng 2 chiều kiểu float có 20 hàng và 30 cột → số phần tử của mảng này là  $20 \times 30 = 60$ . (mỗi phần tử của mảng là một số thực với độ chính xác đơn).

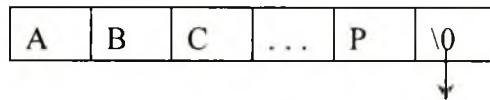
*Ví dụ 1.15:* int m[][] = {{5, 2, -3, 1}, {-2, 4, -7, 3}}; ← mảng 2 chiều này có 2 hàng, 4 cột.

Về phương diện đại số ta có thể coi mảng 2 chiều kích thước mxn là một ma trận chữ nhật cấp mxn. Với Vd1.15, ta có biểu diễn đại số của mảng m[][] như sau:

```
5 2 -3 1
-2 4 -7 3
```

**Kiểu xâu ký tự (String).** String có cấu trúc như mảng một chiều có kiểu char (mỗi phần tử là một ký tự). Song khác cơ bản với mảng các ký tự là xâu ký tự có phần tử cuối là ký tự “kết thúc” xâu, đó là ký tự NULL hay ‘\0’. (Xem hình 1.3).

Chỉ số các phần tử thuộc xâu i = 0 1 2 ..... L (L: độ dài xâu)



Ký tự kết thúc xâu

Hình 1.3. Cấu trúc xâu ký tự

- *Cú pháp khai báo string. Có 3 dạng tương đương:*

```
char xau1[7]={'O','P','Q','R','S','T'};
```

```
char xau2[7]="OPQRST";
```

```
char xau3[]="OPQRST";
```

- *Khởi trị cho string: xau1="\0";*

**Lưu ý:** Như trên đã nói, dữ liệu kiểu string có một ô nhớ cuối cùng chứa ký tự báo kết thúc xâu, vì thế mà trong khai báo xau1, xau2 trên đây, độ dài 2 xâu này phải là 7 (để trong 2 ngoặc vuông).

+ **Kiểu cấu trúc và hợp (Struct và Union, có thể xem là bản ghi như của Pascal, Visual Foxpro).**

Hai kiểu dữ liệu có cấu trúc này khai báo như nhau bởi một cú pháp như nhau.

Điều khác nhau duy nhất giữa chúng là:

- Mỗi thành phần của struct dùng một vùng nhớ **riêng biệt**.
- Còn các thành phần của union dùng **chung một vùng nhớ**.

Cú pháp khai báo. Có các dạng:

+ Dạng chuẩn 1: → Có thể không cần tên ở đây!

struct <tên cấu trúc>

{ kiểu <thành phần 1>;

kiểu <thành\_phần\_k>;

} <tên\_biến\_cấu\_trúc>;

+ Dạng chuẩn 2:

```
struct <tên_cấu_trúc>
```

{ **kiểu** <thành\_phần\_1>;

kiểu <thành phần k>;

< tên mảng cấu trúc [n]

± Dang chuẩn 3:

```
struct stem
```

### {kiểu thành phần}

kiểu <thành phần>:

—  
—

10.  $\frac{3}{4} \times 1 = \frac{3}{4}$

kiểu (nhóm\_phan\_k),

), < - Cho hay khong co ten bien. (Hin: luon co chain phay)

+ Khai báo nhỡ từ khóa typedef:

type def struct <ten\_cau\_truc> ← không nhất thiết có tên\_cáu\_trúc

```
{    kiểu <thành_phần_1>;
```

kiểu <thành\_phần\_2>;

• • • • • • • • • • •

kiểu <thành\_phần\_k>;

*Ví dụ 1.16:* Cấu trúc sau đây quản lý 50 nhân viên, với các mục (các thành phần): name, job, phone, add.

```
struct nv
{ char name[18];
  char job[30];
  long phone ;
  char add[30];
} a[50];
```

- Cú pháp khai báo **cấu trúc lồng nhau**. Với nhiều sinh viên, cấu trúc này hay bị khai báo sai sót nhất. Để đọc giả dễ tiếp thu mục này, trước tiên ta hãy trình bày trên một ví dụ cụ thể:

*Ví dụ 1.17:* Khai báo một cấu trúc chứa các cấu trúc con:

- + Cấu trúc quản lý điểm thi 3 môn toán, lý, hóa.
- + Cấu trúc quản lý ngày, tháng, năm sinh của thí sinh
- + Cấu trúc mẹ (chứa 2 cấu trúc trên) quản lý các mục: họ tên, điểm thi 3 môn trên; ngày, tháng, năm sinh và tổng điểm thi 3 môn đó.

Để đáp ứng yêu cầu ở ví dụ này, ta khai báo các struct như sau:

```
struct diemthi
{ float t,l,h; } ;

struct ngaysinh
{ int d,m,y; } ;

struct sinhvien
{ char hoten[25];
  struct diemthi dt;
  struct ngaysinh ns;
  int tg;
} sv[100];
```

+ *Từ ví dụ dễ hiểu trên đây ta có thể tổng quát hóa khai báo cấu trúc lồng nhau*

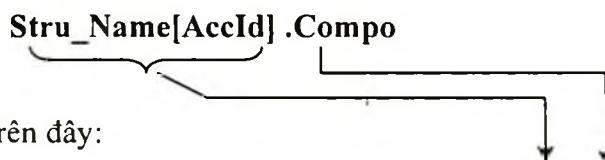
```
struct <cautrl>
{ <các thành phần>; } ; // Cú pháp khai báo các thành
struct <cautr2>           phần như trên //
{ <các thành phần>; };
struct <cautr_me>
{<các thành phần của cautr_me>;
struct cautr1 th_phan_tuong_ung; ← bắt buộc phải có từ
                           khoá struct
struct cautr2 th_phan_tuong_ung;
} ;
```

+ *Cách truy cập vào mỗi phần tử của cấu trúc.*

Một vài từ viết tắt:

- Tên biến cấu trúc: **Stru\_Name**;
- Chỉ số của phần tử cần truy cập: **AccId**;
- Thành phần của cấu trúc mẹ: **MainStruCompo**;
- Thành phần của cấu trúc con: **SubStruCompo**;
- Thành phần tương ứng: **Compo**.

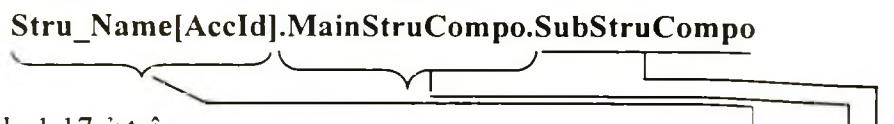
- **Cấu trúc đơn.** Việc truy cập vào một phần tử của cấu trúc đơn theo quy cách sau:



Trở lại Ví dụ 1.16 trên đây:

```
printf("\nNhập tên nhân viên thứ 3:"); scanf("%s", a[3].name);
```

- **Cấu trúc lồng nhau.** Để truy cập vào một phần tử trong cấu trúc lồng nhau ta làm theo quy cách sau:



Trở lại Ví dụ 1.17 ở trên.

```
printf("\nXem năm sinh của thí sinh thứ 51:"); printf("d%", sv[51].ns.y);
```

### Kiểu dữ liệu tự định nghĩa

Rất nhiều bài toán thực tế cho thấy chỉ với các kiểu dữ liệu mà C cung cấp sẵn không đủ để mô tả các đối tượng của những bài toán này. Khi đó người dùng phải tự định nghĩa kiểu. Có hai từ khóa giúp ta làm việc này.

- Định nghĩa kiểu liệt kê nhờ từ khóa enum với cú pháp chung sau đây:

```
typedef enum Tên_kiểu {Tênđốitượng_1, Tênđốitượng_2, . . . ,  
Tênđốitượng_n} Tên_bieln
```

*Ví dụ 1.18:* `typedef enum {True, False} bool;`

Đặt tên mới cho kiểu dữ liệu đã tồn tại nhờ từ khóa typedef

```
typedef <kiểu_có_sẵn_của_C> <tên_mới>
```

*Ví dụ 1.19:* `typedef float chieu_cao;`

```
chieu_cao nha_tu_nhan; //Nhà tư nhân
```

### Dữ liệu kiểu tệp

- Ngôn ngữ C

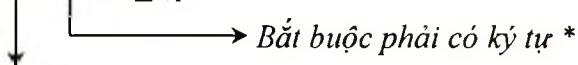
Có hai loại tệp:

+ *Tệp văn bản (Text file)*: Đó là kiểu tệp lưu các ký tự (trừ các ký tự điều khiển) thuộc bảng mã ASCII theo từng dòng và chứa các ký tự kết thúc dòng là CR (mã 3 hoặc '\r') và LF (mã 10 hoặc '\n'). Ngoài ra tệp văn bản còn chứa ký tự ^Z (mã 26) biểu thị kết thúc tệp.

+ *Tệp nhị phân (Binary file, còn gọi là tệp Mã máy)*: Đó là kiểu tệp chỉ chứa các dữ liệu dạng nhị phân (mã máy) tổ chức theo từng cấu trúc (*không tổ chức theo dòng*). Trong tệp nhị phân có cả các mã điều khiển và chỉ có ký tự LF ('\n') mà không có ký tự CR ('\r').

+ Nhắc lại một số lệnh cơ bản xử lý tệp

- Khai báo biến kiểu tệp: FILE \*biến\_tệp;



Từ khóa này bắt buộc viết HOA

- Mở tệp để ghi dữ liệu lên tệp (tức lên đĩa)

Tệp văn bản: `biến_tệp=fopen(["Đường_dẫn]Tên_tệp", "wt")`

Hoặc không cần ký tự t

Tệp nhị phân: `biến_tệp=fopen(["Đường_dẫn]Tên_tệp", "wb")`

(Lưu ý: ký hiệu [ ] là *tùy chọn (options)*)

- Mở tệp để đọc dữ liệu từ tệp (tức đọc từ đĩa)

Tệp văn bản: biến\_tệp=fopen(["Đường\_dẫn]Tên\_tệp","rt")

Tệp nhị phân: biến\_tệp=fopen(["Đường\_dẫn]Tên\_tệp","rb")

- Việc ghi/đọc tệp trong C: Độc giả xem lại các giáo khoa về Lập trình với ngôn ngữ C.

*Ví dụ 1.20:* Giả dụ thư mục TC cát ở ổ đĩa D, khi ấy lệnh mở một file với tên MyFile.Txt để ghi dữ liệu lên nó là:

```
FILE *f;
f=fopen("D:\\TC\\BIN\\ MyFile.Txt","w")
```

- Ngôn ngữ C<sup>++</sup>:

+ Phân loại tệp và khai báo biến tệp trong C<sup>++</sup> giống hệt như ở C

+ Điều khác cơ bản với C là: trong C<sup>++</sup> trước tiên phải khai báo hàm tiêu đề "fstream.h"- đó là lớp cơ sở chứa một vài lớp khác liên quan tới thao tác trên file.

+ Mở tệp để đọc dữ liệu với cú pháp:

```
ifstream fi(fname);
```

Trong đó fi là *đối tượng của lớp ifstream* dùng để **đọc** dữ liệu ở tệp có tên fname. fname có thể là **hằng xâu ký tự** khai báo trước bởi #define hoặc là một xâu ký tự bị kẹp giữa hai dấu ngoặc kép “ ” ghi đầy đủ, chính xác đường dẫn và tên tệp ở máy tính bạn đang làm việc

*Ví dụ 1.21:* #define fname "D:\\TC\\BIN\\MyFile.In"

Fstream fi(fname) hoặc là fstream fi("D:\\TC\\BIN\\MyFile.In")

Trong đó fi là *đối tượng để đọc* tệp.

fi>>n; ← đọc số phần tử n từ file MyFile.In

- Đóng tệp: fi.close();

- Mở tệp để xuất dữ liệu hoặc ghi dữ liệu theo syntax: ofstream fo(fname)

Trong đó fo là *đối tượng để xuất dữ liệu* từ tệp hoặc ghi lên tệp.

*Ví dụ 1.22:*

ofstream fo ("D:\\TC\\BIN\\MyFile.Ou")

Ghi dữ liệu lên tệp:

```
char *Ht="Steve Job";
fo<<Ht; ← Ghi xâu ký tự Ht vào dòng hiện tại của
tệp (dòng đang có con trỏ)
char string[20];
cout<<Nhập họ tên:"; gets(string)
fo<<string;
fo.close();
```

#### 1.2.4. Vai trò của cấu trúc dữ liệu trong một chương trình ứng dụng tin học

Về điều này, năm 1970 giáo sư toán học Niklaus Wirth đại học Zuyric Thụy Sĩ đã phát biểu bằng một công thức nổi tiếng, đầy đủ nhất và không thể ngắn gọn hơn:

#### PROGRAMS = DATA STRUCTURE + ALGORITHMS

Trong công thức trên ta cần hiểu dấu “+” là sự *kết hợp hợp lý* giữa data structure và algorithm.

Cho đến bây giờ, chúng ta đều thấy rõ ràng: bất cứ một chương trình ứng dụng dù lớn hay nhỏ; bất cứ một phần mềm ứng dụng nào của tin học cũng là sự kết hợp nhuần nhuyễn giữa cấu trúc dữ liệu và thuật toán. Có hai điều bạn cần nhận thức:

- + Nếu thuật toán sai thì hiển nhiên kết quả chương trình sai. Điều này mọi sinh viên (kể cả sinh viên mới học lập trình) đều dễ nhận biết và đều rõ.

- + Nếu thuật toán **đúng** mà kiểu dữ liệu hoặc cấu trúc dữ liệu **chọn dùng không phù hợp** thì kết quả cũng sai. Điều thứ hai này khá nhiều sinh viên không nhận biết được và không thấm nhuần điều này. Khi thực hành, rất nhiều trường hợp thuật toán đúng mà kết quả vẫn sai do chọn dữ liệu không hợp lý.

Dưới đây là những ví dụ minh chứng cho mối liên kết hữu cơ, chặt chẽ giữa dữ liệu (cấu trúc dữ liệu) và thuật toán.

*Ví dụ 1.23:* Có các loại tờ giấy bạc với các mệnh giá sau: 50 ngàn, 20 ngàn, 10 ngàn, 5 ngàn, 2 ngàn và 1 ngàn. Lập trình tìm cách đổi số tiền St ra các tờ bạc trên sao cho tổng số các tờ bạc là ít nhất. (St nhập từ bàn phím vào và nhập *tròn tới ngàn đồng!*). Bản code sau mô tả đúng thuật toán để giải bài toán trên, song kết quả sai vì chọn kiểu dữ liệu không phù hợp.

```

#include<iostream.h>
#include<conio.h>
main()
{ int st, lt[6], sto[6]; int i; clrscr();
  cout<<"\n Nhập số tiền cần đổi st="; cin>>st;
  lt[0]=50000;
  lt[1]=20000;
  lt[2]=10000;
  lt[3]=5000;
  lt[4]=2000;
  lt[5]=1000;
  for (i=0; i<6; i++)
  { sto[i]=st/lt[i];
    st=st%lt[i];
  }
  i=0;
  cout<<"\n Số tờ 50 ngàn="<<sto[i];
  cout<<"\n Số tờ 20 ngàn="<<sto[i+1];
  cout<<"\n Số tờ 10 ngàn="<<sto[i+2];
  cout<<"\n Số tờ 5 ngàn="<<sto[i+3];
  cout<<"\n Số tờ 2 ngàn="<<sto[i+4];
  cout<<"\n Số tờ 1 ngàn="<<sto[i+5];
  getch();
}

```

Lỗi của chương trình ở đây: dùng kiểu dữ liệu không phù hợp với dữ liệu vào của bài toán đã cho!

Thuật toán **đúng**, Kết quả chạy chương trình vẫn sai! (*Do chọn dữ liệu không phù hợp*)

Như lúc đầu đã nói, sai trong code này là ở chỗ, một sinh viên đã chọn kiểu dữ liệu **int** (tùy khoá in đậm). Khi đọc đầu bài, sinh viên này chỉ chăm chú vào việc tìm thuật toán mà không chú ý đến bộ dữ liệu vào và giá trị cao nhất của các dữ liệu đó: ở bài toán này, giá trị cao nhất là 50000 (vượt quá miền giá trị của kiểu **int**: -32768 → 32767). Bạn chỉ cần sửa lại kiểu dữ liệu là **long int** thì chương trình sẽ cho kết quả đúng.

*Ví dụ 1.24:* Lập trình quản lý việc bán hàng gồm các khoản mục sau: tên hàng, số lượng hàng, tiền bán, tổng số tiền bán, tổng số hóa đơn. Yêu cầu:

- + Chương trình phải tạo được hóa đơn mới và hủy được hóa đơn cần hủy.
- + Mọi thay đổi về tiền bán phải được phản ảnh vào tổng số tiền bán (tức là tổng số tiền bán phải được cập nhật *tự động*).

Bài này “phức tạp” hơn bài trên ở chỗ có nhiều việc cần phải xử lý hơn, tuy nhiên “thuật toán” để làm các tính toán ở bài ra thì dễ hơn nhiều: đọc xong đề bài là bạn đã hình dung ra cách tính tiền bán từng loại hàng và tổng tiền bán hàng. Tuy nhiên nhiều bạn chưa hình dung nổi cách giải quyết 2 yêu cầu của bài toán, đặc biệt là yêu cầu thứ 2 (*tự động cập nhật tổng số tiền bán!*).

Với yêu cầu của bài trên, *hợp lý nhất* và *thuận lợi nhất* cho việc thiết kế chương trình thỏa các yêu cầu của bài này là bạn dùng cấu trúc dữ liệu *class* trong lập trình hướng đối tượng của C++.

Trong class này bạn xây dựng 3 hàm chủ chốt: hàm tạo hóa đơn, hàm hủy hóa đơn và hàm sửa đổi dữ liệu nói chung và tiền bán một mặt hàng nói riêng và tìm cách cập nhật cho tổng tiền bán (Các việc còn lại là dễ dàng hơn). Mọi việc chọn cấu trúc dữ liệu khác đều làm bạn lúng túng, sa lầy trong khi thiết kế chương trình và có thể thất bại.

*Ví dụ 1.25:* Nếu thầy giáo giao cho bạn bài toán: “*Viết chương trình nhập vào một số nguyên dương ở hệ đếm tùy ý và đổi nó sang hệ đếm mới có cơ số bất kỳ và ngược lại*” thì bạn nghĩ gì về việc *chọn kiểu dữ liệu* cho bài toán không đơn giản này?

Như các bạn thấy: Bài toán là tổng quát. Giả dụ, trước hết bạn chỉ muốn thử viết chương trình để đổi một số nguyên dương bất kỳ ở một trong các hệ đếm quen thuộc là thập phân sang hệ nhị phân (và không có chiều ngược lại!)

Với 2 hệ đếm này, thuật toán để làm việc trên bạn đã biết và thậm chí đã thành thạo từ hồi còn học THPT và bạn dễ dàng viết code thành công. Nhưng với dạng tổng quát của bài toán thì chắc chắn đó là việc không dễ dàng với khá nhiều sinh viên! Vấn đề gây khó với bạn ở tình huống tổng quát không chỉ là ở thuật toán mà còn là bạn có biết chọn cấu trúc dữ liệu phù hợp với yêu cầu của bài toán hay không! Giả sử về mặt toán học bạn đã biết mười mươi cách chuyển một số tùy ý từ hệ đếm này sang hệ đếm khác! Nhưng nếu bạn chọn cấu trúc dữ liệu không phù hợp chắc chắn bạn sẽ gặp rất nhiều khó khăn với việc viết code cho bài toán này. Với kinh nghiệm của người đi trước nhiều năm, xin khuyến cáo bạn nên chọn dữ liệu là *xâu ký tự để lưu các digits (chữ số)* và *kết quả của chương trình*. Với kiểu dữ liệu này việc viết chương trình của bạn sẽ

nhé nhàng hơn; bản code của bạn sáng sủa hơn, gọn hơn, chạy nhanh hơn. Bạn sẽ thấy điều này ngay sau khi bạn đọc và test bản code dưới đây của bài toán nêu ở ví dụ 1.25.

*Bài toán Đổi hệ Tổng quát “Đổi một số ở hệ đếm cơ số M sang hệ đếm cơ số N và ngược lại”.*

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<conio.h>
char *doi_he(char *number, int m, int n)
{
    static char ketqua[17];
    char chuso[16] = "0123456789ABCDEF";
    int i = 0, giatri = 0, len;
    len = strlen(number);
    while(i<len)
    {
        giatri = giatri * m + (strchr(chuso,
            number[i]) - chuso);
        i++;
    }
    i=16;
    ketqua[17] = 0;
    do
    {
        ketqua[i] = chuso[giatri % n];
        giatri /= n;
        i--;
    } while (giatri > 0);
    return (ketqua + i + 1);
}
```

```
}

void main()
{ int m, n; char number[17], *ketqua; clrscr();
    do
    { cout<<"\n Nhập Cơ số M của hệ đếm xuất
        phát (2 - 16):";
        cin>>m;
    }
    while (m<2 || m>16);
    cout<<"\n Nhập số nguyên thuộc hệ đếm cơ số
        "<<m<<":";
    gets(number);
    do
    {
        cout<<"Nhập Cơ số N của hệ đếm mới (2 - 16):";
        cin>>n;
    } while (n<2 || n>16);
    cout<<" Số "<<number<<" o hệ đếm cơ số "<<m;
    cout<<" biểu diễn o hệ đếm cơ số "<<n<<" có giá
        trị là:";
    ketqua = doi_he(number, m, n);
    puts(ketqua);
    getch();
}
```

## BÀI TẬP CHƯƠNG 1

### I. Xây dựng thuật toán bằng ngôn ngữ tự nhiên hoặc mã giả

Dùng ngôn ngữ tự nhiên hoặc mã giả diễn đạt thuật toán các bài toán sau:

1.1. Tìm ước số chung lớn nhất của hai số tự nhiên nhờ thuật toán Euclidean dùng phép chia liên tiếp.

1.2. (Trò chơi bốc sỏi sơ đẳng). Có 26 viên sỏi, hai người chơi lần lượt bốc sỏi, đến lượt mỗi người bốc nhiều nhất là 4 viên, Cứ bốc cho tới khi hết sỏi. Người nào phải bốc viên cuối cùng là người thua cuộc. Hãy tìm một chiến thuật chơi giữa người và máy tính sao cho máy tính **bốc sau nhưng luôn luôn thắng**.

1.3. (Trò chơi bốc sỏi “đẳng cấp 1”-Level 1). Có N viên sỏi, hai đội thủ A, B tham gia chơi bốc sỏi. A bốc trước. Mỗi đội thủ đến lượt bốc ít nhất 1 viên, nhiều nhất là  $N/2$  viên. Đội thủ nào đến lượt không còn sỏi để bốc thì thua. Giả sử hai đội thủ đều chơi rất giỏi. Cho biết A thắng hay thua?

1.4. (Trò chơi bốc sỏi “đẳng cấp 2”-Level 2) Có 2 đồng sỏi với số viên lần lượt là M, N. hai đội thủ tham gia chơi: A luôn đi trước. Luật chơi: đến lượt, mỗi đội thủ chọn đồng sỏi tùy ý, bốc ít nhất 1 viên, tối đa cả đồng. Đội thủ nào đến lượt mình mà không đi nổi thì thua. Giả sử hai đội thủ đều chơi rất giỏi. Xác định A thắng hay thua?

1.5. (Trò chơi NIM- bốc sỏi “đẳng cấp 3”-Level 3) Có N đồng sỏi, hai đội thủ A, B tham gia chơi bốc sỏi luân phiên (lần lượt đan xen nhau); đến lượt, mỗi đội thủ được bốc ít nhất 1 viên và tối đa cả đồng sỏi đã chọn. Đội thủ nào đến lượt mà không đi nổi thì thua. Giả sử A luôn đi trước và hai người đều chơi rất giỏi. Xác định A thắng hay thua?

1.6. (**Tập Nửa Bội-Multi\_Half**) Tập Nửa Bội A là tập được định nghĩa như sau:

a.  $a \in A$ ;

b. Nếu  $k \in A$  thì  $2k+1 \in A$  và  $3k+1 \in A$ . Ngoài ra tập A không chứa bất kỳ số nào khác.

In ra n số đầu tiên của A ( $n < 100$ ) theo trật tự tăng dần. Ví dụ dãy tăng dần gồm 9 số đầu tiên của A là: 1, 3, 4, 7, 9, 10, 13, 15, 19, ...

1.7. (*Tam giác ký tự Đối xứng*) Cho trước xâu lý tự “abcde.....xyz”. Hãy in lên màn hình tam giác gồm các ký tự đã cho với cấu trúc sau:

```

      a
      bcd
      cdedc
      defgfed
      -----

```

mno-----vxyzxyzv-----onm

1.8. Tìm một chiến lược phân tích một số tự nhiên thành tổng của các số tự nhiên khác sao cho tích của các số hạng thuộc tổng đó là lớn nhất.

1.9. Tìm các số siêu nguyên tố có số lượng các chữ số không quá 10. Một số được gọi là siêu nguyên tố nếu sau khi xóa một số tùy ý các chữ số kế tiếp ở bên phải thì phần còn lại của số đã cho vẫn là một số nguyên tố. Ví dụ: cho số 23: xóa 3 thì 2 cũng là số nguyên tố; 137: xóa 7 thì phần còn lại là 13 cũng là số nguyên tố.

## II. Dùng Flowchart thiết kế thuật toán cho các bài toán sau

1.10. Xây dựng flowchart cho bài toán 1.2

1.11. Tìm tất cả các nghiệm nguyên dương có thể có của phương trình:  $3x+4y-5z=9$  với  $0 \leq x,y,z \leq 50$ .

1.12. Cho trước mảng các số (thực hoặc nguyên) tính tổng các số trong mảng.

1.13. Tìm ước số chung lớn nhất của 2 số nguyên dương bằng thuật toán Euclide dùng phép chia liên tiếp.

1.14. Tính hàm  $e^x$  với giá trị x và sai số  $\epsilon$  cho trước nhờ khai triển Taylor:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!}$$

1.15. Thực hiện phép chia hai số nguyên dương a, b bằng phép trừ để tính dư và thương của chúng.

1.16. Tìm và in lên màn các số hoàn hảo. (Số hoàn hảo là số bằng tổng các ước của nó. Ví dụ: 6 là số hoàn hảo vì  $6=1+2+3$ ).

1.17. Cặp số tự nhiên ( $n, n^2$ ) gọi là cặp số Lucas nếu  $n$  trùng với phần cuối của  $n^2$ . Ví dụ (6, 36); (25, 625) là các cặp Lucas. Tìm các cặp Lucas không vượt quá số n cho trước mà  $n < 2147483638$ .

1.18. Phép tính  $n!!$  được định nghĩa như sau:

$$n!! = \begin{cases} 1.3.5...n & (n: lẻ) \\ 2.4.6...n & (n: chẵn) \end{cases}$$

1.19. Cho số tự nhiên  $n$ . Tính:

$$\frac{1}{1 + \frac{1}{3 + \frac{1}{\dots + \frac{1}{2n-1 + \frac{1}{2n+1}}}}}$$

1.20. Cho số nguyên dương  $n$ . Hãy liệt kê mọi cách có thể có để phân tích  $n$  thành tổng các số nguyên dương *liền tiếp*. Ví dụ  $n=9$  ta có  $9=2+3+4$  là một phương án.

1.21. Gọi các số “Tam tam” là những số có 3 chữ số sao cho khi đảo ngược các chữ số của nó thì được số nguyên tố với số đã cho. Ví dụ 761 là một số như vậy vì 167 nguyên tố cùng nhau với 761.

1.22. Số siêu nguyên tố là một số nguyên tố mà sau khi xoá đi một số tùy ý các chữ số kể từ cận phải của nó thì phần còn lại vẫn là số nguyên tố. Ví dụ: 719333 là một số như vậy! Vì 71933, 7193, 719, 71 và 7 đều là các số nguyên tố. Tìm và in lên màn hình mọi số siêu nguyên tố có  $N$  chữ số (với  $N<9$ ).

**III. Dùng ngôn ngữ lập trình bậc cao C/C++ diễn đạt thuật toán (viết code) cho các bài toán sau**

1.23. Bài 1.2

1.24. Bài 1.5

1.25. Bài 1.6

- 1.26. Bài 1.7
- 1.27. Bài 1.8
- 1.28. Bài 1.9
- 1.29. Bài 1.10
- 1.30. Bài 1.16
- 1.31. Bài 1.15
- 1.32. Bài 1.18
- 1.33. Bài 1.20
- 1.34. Bài 1.21
- 1.35. Bài 1.22.

#### IV. Đánh giá độ phức tạp (ĐPT) của hàm

1.36. Đánh giá  $O(f(x))$  của hàm  $f(x)$ :  $f(x)=(x+1)\log(x^2+1)+3x^2$

1.37. ĐPT của các hàm sau có là  $O(x^2)$  không?

- a.  $f(x)=x^2+2012$
- b.  $f(x)=x\log x$
- c.  $f(x)=\frac{x^4}{2}$
- d.  $f(x)=2^x$

1.38. Chứng minh rằng: ĐPT của:

- a.  $2^x+2012$  là  $O(3^x)$
- b.  $\frac{x^3+2x}{2x+1}$  là  $O(x^2)$

1.39. Tìm một số nguyên dương nhỏ nhất n sao cho ĐPT của các  $f(x)$  dưới đây là  $O(x^n)$ :

- a.  $2x^3+x^2\log x$
- b.  $\frac{x^5+5\log x}{x^4+1}$
- c.  $3x^5+(\log x)^4$
- d.  $\frac{x^4+x^2+1}{x^4+1}$

1.40. Cho một đánh giá tốt nhất với O\_Big cho các hàm dưới đây (tức hãy chọn các hàm đơn giản có bậc thấp nhất):

- a.  $(n\log n+n^2)(n^3+2)$
- b.  $(n!+2^n)(n^3+\log(n^2+1))$
- c.  $(n^3+n^2\log n)(\log n+1)+(17\log n+19)(n^3+2)$
- d.  $(2^n+n^2)(n^3+3^n)$
- e.  $(n\log n+1)^2+(\log n+1)(n^2+1)$
- f.  $n^{2^n}+n^{n^2}$

1.41. Cho hàm  $f(x)$  có  $O(g(x))$ ,  $g(x)$  có  $O(h(x))$ . Chứng tỏ rằng  $f(x)$  có ĐPT là  $O(h(x))$

- 1.42. Cho  $f(x)$  có  $O(g(x))$ . Từ đó có thể suy ra  $2^{f(x)}$  có  $O(2^{g(x)})$  không?
- 1.43. a. Chứng minh rằng  $f(n)=n\log n$  có ĐPT là  $O(\log n!)$
- b. Cho  $H_n=1+\frac{1}{2}+\frac{1}{3}+\dots+\frac{1}{n}$ . Chứng minh rằng ĐPT của  $H_n$  là  $O(\log n)$ .

#### V. Đánh giá ĐPT một vài thuật toán

Đánh giá Big\_O của một số bài toán và thuật toán sau:

1.44. Các bài toán: 1.2; 1.5; 1.6; 1.10; 1.16; 1.18; 1.20

1.45. Thuật toán tìm kiếm Tuần tự (Sequential Search)

1.46. Thuật toán sắp xếp Nhanh (Quick Sort)

1.47. Thuật toán sắp xếp Vun đống (Heap Sort)

#### VI. Sử dụng một vài kiểu dữ liệu cơ bản có cấu trúc

1.48. Ta gọi điểm Yên ngựa trong ma trận chữ nhật kích thước mxn là phần tử có giá trị **nhỏ nhất trên hàng và lớn nhất trên cột**. Lập trình tìm điểm Yên ngựa trong một ma trận mxn cho trước. Ví dụ trong ma trận 3x4 sau đây, điểm Yên ngựa là phần tử A[1,1]=4:

1 3 5 9

(Lưu ý: Trong C/C++, mặc định, chỉ số của các phần tử

5 4 5 6

thuộc mảng được đánh số từ 0!)

7 1 6 2

1.48. Hãy chọn một cấu trúc dữ liệu hợp lý để quản lý 3 thông tin sau của khách gửi tiết kiệm: mã tài khoản, họ tên, số tiền gửi (không lưu lên đĩa). Lập trình tìm một mã tài khoản cho trước. Yêu cầu:

a. Khai báo một **hàng mảng** có kích cỡ tối đa là 300 để lưu 3 thông tin trên của khách.

b. Dùng binary search giải bài toán theo yêu cầu đã nêu.

1.50. Một kỳ thi đại học khối A, các thí sinh phải thi 3 môn: Toán, Lý, Hóa. Hãy chọn một cấu trúc dữ liệu phù hợp để quản lý những thông tin sau của các thí sinh: ngày, tháng, năm sinh; điểm Toán, điểm Lý, điểm Hóa, tính điểm tổng của các thí sinh. Hãy lọc lên màn hình những thí sinh có tổng điểm (cả 3 môn)  $> 15$  theo form sau:

Họ tên	Ngày sinh	Điểm tổng
Khanh	21-7-1997	23.75

(dấu phân cách các dữ liệu ngày sinh là ký tự : “-“)

Các dữ liệu không lưu lên đĩa.

1.51. Cần quản lý các sinh viên của một trường đại học gồm các thông tin sau: Mã sinh viên; Họ tên, Ngày, tháng, năm sinh; Giới tính; Lớp (ví dụ QTKD12); Quê. **Các thông tin cần lưu lên đĩa.** Lập trình thực thi các việc sau:

- 1- Nhập liệu (Data input)
- 2- Xem toàn danh sách
- 3- Tìm & hiển thị một lý lịch
- 4- Xóa một lý lịch theo mã
- 5- Thoát

**Yêu cầu:** Khi nhập xong các thông tin của một sinh viên, phải in lên màn hình câu hỏi “Có nhập tiếp không?”. Ngừng nhập gõ ký tự “k” (Gõ “c” hoặc “C” để nhập tiếp). Cũng yêu cầu như vậy với các thao tác tìm và xóa 1 lý lịch. (Thao tác xóa dữ liệu là rất quan trọng bởi vậy khi thiết kế chương trình các bạn nên có câu vấn đáp như trên để người dùng tránh “xóa nhầm” những đối tượng đang quản lý).

### GỢI Ý HOẶC ĐÁP ÁN

#### I. Xây dựng thuật toán bằng ngôn ngữ tự nhiên hoặc mã giả

1.2. (Bài bốc sỏi “sơ đẳng”). Nếu không đội thủ nào biết được số sỏi đội thủ của mình vừa bốc thì theo lý thuyết xác suất khả năng thua của hai đội thủ là như nhau. Còn nếu đội thủ đi sau mà biết được số sỏi vừa bị bốc bởi đội thủ của mình là N viên thì người này chỉ cần bốc ( $5-N$ ) sẽ chắc thắng. Vì cứ sau mỗi lần bốc thì tổng số sỏi được bốc bởi cả hai là  $N+(5-N)=\text{const}$ . Theo bài ra tổng số sỏi là  $26=5*5+1$ . Như vậy, sau 5 lần bốc thì số sỏi còn lại là 1 và đội thủ đi trước phải bốc viên này nên *luôn chắc thua!*

1.5. (Trò chơi NIM) Ký hiệu số viên sỏi của các đống là  $S_1, S_2, \dots, S_N$  Ký hiệu  $\wedge$  là phép *tuyển loại trừ* và  $S=S_1 \wedge S_2 \wedge \dots \wedge S_N$  ( $\wedge$  chính là phép XOR xử lý bit trong C/C<sup>++</sup>). Ta nhắc lại một số luật cơ bản của *tuyển loại trừ*:

+ Phép *tuyển loại trừ*  $a \wedge b$  chỉ nhận giá trị 1 khi  $a \neq b$  (trong mọi trường hợp còn lại *tuyển loại trừ* đều có giá trị 0).

- +  $a \wedge 0=a$ .
- +  $a \wedge a=0$  (Luật “Nuốt”)
- +  $a \wedge 1= ! a$  (! a: phủ định a)
- +  $a \wedge b \wedge a=b$

+ Suy rộng tính chất cuối cùng: Trong biểu thức chỉ chứa phép  $\wedge$  thì có thể xóa đi số chẵn các toán hạng giống nhau thì kết quả của biểu thức ấy không hề thay đổi.

Dưới đây ta sẽ chứng tỏ *bất biến thua* của trò chơi NIM là  $S=0$ . Nói rõ hơn: nếu  $S=0$  thì đến lượt đối thủ nào đi thì người ấy sẽ thua. Bất biến vừa nói có nghĩa như sau: Nếu biểu diễn các số sói  $S_1, \dots, S_N$  bằng các số nhị phân trong một bảng thì số lượng các bit 1 ở mọi cột đều là số chẵn. Dưới đây là một ví dụ cụ thể:

Đóng sói	Dạng nhị phân			
$S_1=13$	1	1	0	1
$S_2=14$	1	1	1	0
$S_3=6$	0	1	1	0
$S_4=7$	0	1	1	1
$S_5=2$	0	0	1	0
$\wedge S = 0$	0	0	0	0

Nếu  $S$  là *tuyển loại trừ* của các  $S_i$ ,  $i=1, \dots, N$ . Với mỗi  $i=1, \dots, N$  ta gọi  $K(i)$  là *tuyển loại trừ khuyết i* với các tính chất sau:

$K(i) = S_1 \wedge S_2 \wedge \dots \wedge S_{i-1} \wedge S_{i+1} \wedge \dots \wedge S_N$ . Nghĩa là  $K(i)$  là *tuyển loại trừ* của các  $S_j$  sau khi đã xóa bỏ  $S_i$ . Còn chính  $S$ , là *tuyển loại trừ* đù của các  $S_i$ , với  $i=1, \dots, N$ .

Do luật “Nuốt”  $S_i \wedge S_i = 0$  và  $a \wedge 0 = a$  với  $\forall a$  nên  $K(i) = S \wedge S_i$ . Ký hiệu  $K(0)$  là  $\wedge$  đù của các  $S_i$ ,  $i=1, \dots, N$ .

Các tổng khuyết của bảng trên là:

- $K(0) = S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge S_5 = 13 \wedge 14 \wedge 6 \wedge 7 \wedge 2 = 0;$
- $K(1) = S_2 \wedge S_3 \wedge S_4 \wedge S_5 = 14 \wedge 6 \wedge 7 \wedge 2 = 13;$
- $K(2) = S_1 \wedge S_3 \wedge S_4 \wedge S_5 = 13 \wedge 6 \wedge 7 \wedge 2 = 14;$
- $K(3) = S_1 \wedge S_2 \wedge S_4 \wedge S_5 = 13 \wedge 14 \wedge 7 \wedge 2 = 6;$
- $K(4) = S_1 \wedge S_2 \wedge S_3 \wedge S_5 = 13 \wedge 14 \wedge 6 \wedge 2 = 7;$
- $K(5) = S_1 \wedge S_2 \wedge S_3 \wedge S_4 = 13 \wedge 14 \wedge 6 \wedge 7 = 2;$

Ta có một số quy luật đáng chú ý sau:

**Mệnh đề 1:** Cho  $S$  là *tuyến loại trừ* của  $N$  số tự nhiên. Khi ấy  $K(i) = S \wedge S_i$ ,  $i=1, 2, 3, \dots, N$ . Nghĩa là muốn hủy một toán hạng trong  $\wedge$  ta chỉ việc thêm chính số hạng ấy vào  $\wedge$ . Cá biệt khi  $S=0$  ta có  $K(i)=S_i$  với  $i=1, 2, 3, \dots, N$ .

**Chứng minh:** Ta đã biết  $S$  là *tuyến loại trừ* của các  $S_i$ ,  $i=1, 2, 3, \dots, N$ . Dùng luật giao hoán và “Nuốt” của  $\wedge$  ta có:

$$S \wedge S_i = (S_1 \wedge S_2 \wedge \dots \wedge S_{i-1} \wedge S_{i+1} \wedge \dots \wedge S_N) \wedge (S_i \wedge S_i) = K(i) + 0 = K(i) \rightarrow$$

Chứng minh xong

**Mệnh đề 2:** Nếu  $x \neq 0$  thì tồn tại cách đi để biến đổi  $S=0$ .

**Chứng minh:** Do  $S \neq 0$  nên ta xét bit 1 trái nhất của biểu diễn nhị phân của  $S=(s_m, s_{m-1}, \dots, s_0)$  với  $s_j=1, s_i=0, i>j$ . Vì  $S$  là *tuyến loại trừ* của các  $S_i$ ,  $i=1, 2, 3, \dots, N$ , nên tồn tại một  $S_i=(a_m, a_{m-1}, \dots, a_0)$  để bit  $a_j=1$ . Ta chọn đồng sói  $S_i$  này. Khi đó ta tính được  $K(i) = S \wedge S_i = (s_m \wedge a_m, s_{m-1} \wedge a_{m-1}, \dots, s_0 \wedge a_0) = (b_m, b_{m-1}, \dots, b_0)$  ở đây  $b_i=s_i \wedge a_i, 0 \leq i \leq m$ .

Ta xét bảng dưới đây:

Đồng sói	Dạng nhị phân			
	$s_3$	$s_2$	$s_1$	$s_0$
$S_1=12$	1	1	0	0
$S_2=14$	1	1	1	0
$S_3=6$	0	1	1	0
$S_4=3$	0	0	1	1
$S_5=2$	0	0	1	0
$\wedge S = 5$	0	1	0	1

- + Tại các cột  $i>j$  thì  $b_i=a_i$  vì  $b_i=s_i \wedge a_i=0 \wedge a_i=a_i$ .
- + Tại cột  $j$  ta có  $b_j=0$  vì  $b_j=s_j \wedge a_j=1 \wedge 1=0$ .
- + Do  $a_j=1, b_j=0$  và  $\forall$  vị trí  $i>j$  đều có  $b_i=a_i \rightarrow S_i>K(i)$ . Nếu thay  $S_i$  bởi  $K(i)$  ta có *tuyến loại trừ* của  $a$  là:

$a = (S \wedge S_i) \wedge K(i) = K(i) \wedge K(i) = 0$  nghĩa là nếu ta bóc ở đồng i đi v viên  $= S_i - K(i)$  thì số sỏi còn lại trong đồng ấy là  $K(i)$  và khi đó *tuyến loại trừ* = 0. Chứng minh xong!

**Mệnh đề 3:** Nếu  $S = 0$  và còn đồng sỏi khác 0 thì mọi cách đi hợp lệ đều dẫn đến  $S \neq 0$ .

*Chứng minh:*

Cách đi hợp lệ là cách đi làm giảm số sỏi của một đồng  $S_i$  ( $1 \leq i \leq N$ ) duy nhất nào đó. Giả sử đồng được chọn là  $S_i = (a_m, a_{m-1}, \dots, a_0)$ . Do  $S_i$  đã thay đổi nên chắc chắn có một bit nào đó bị đảo:  $1 \rightarrow 0$  hoặc  $0 \rightarrow 1$ . Ký hiệu bit bị đảo ấy là  $a_j$ . Khi đó tổng số bit 1 trên cột  $j$  sẽ tăng hoặc giảm 1 đơn vị và vì vậy tổng này sẽ không còn là số chẵn. Từ đó suy ra rằng bit  $j$  trong  $S$  sẽ là 1 tức  $S \neq 0$ . Chứng minh xong!

Phản lập luận chủ yếu trong chứng minh mệnh đề 2 nhằm mục đích chỉ ra sự tồn tại của một tập  $S_i$  thỏa mãn tính chất  $S_i > S \wedge S_i$ . Nếu tìm được một  $S_i$  đối thủ đến lượt đi sẽ bốc  $S_i$  ( $S \wedge S_i$ ) viên ở đồng sỏi  $i$ .

Sau đây ta sẽ viết hai hàm diễn đạt thuật toán trò chơi NIM. Ta dùng mảng  $S[1..N]$  kiểu nguyên chứa số sỏi của mỗi đồng.

Trước hết ta viết hàm Result trả về giá trị của *tuyến loại trừ* của các đồng sỏi. Như vậy khi  $x=0$  thì người nào đi trước sẽ thua, ngược lại khi  $S \neq 0$  sẽ thắng.

```
int Result ()
{
    int S=0;
    for(int i=0; i< S.Len; ++i) /*S: độ dài của mảng S, rõ
                                   hơn là số sỏi của đồng S*/
        S ^= S[i];
    return S;
}
```

Dưới đây là hàm mô tả cách đi Way().

Hàm này gọi hàm Result để xác định giá trị  $S$ . Nếu  $S=0 \rightarrow$  thua thì chọn một đồng còn sỏi, chẳng hạn đồng *còn nhiều sỏi nhất* để bóc tạm 1 viên nhằm kéo dài cuộc chơi. Nếu  $S=0$  và các đồng khác đều hết sỏi thì đương nhiên thua. Trường hợp  $S \neq 0$  thì tìm chiến thuật đi chắc thắng như sau:

+ Bước 1: Tìm đồng sỏi  $i$  thỏa mãn hệ thức  $S_i > S \wedge S_i$

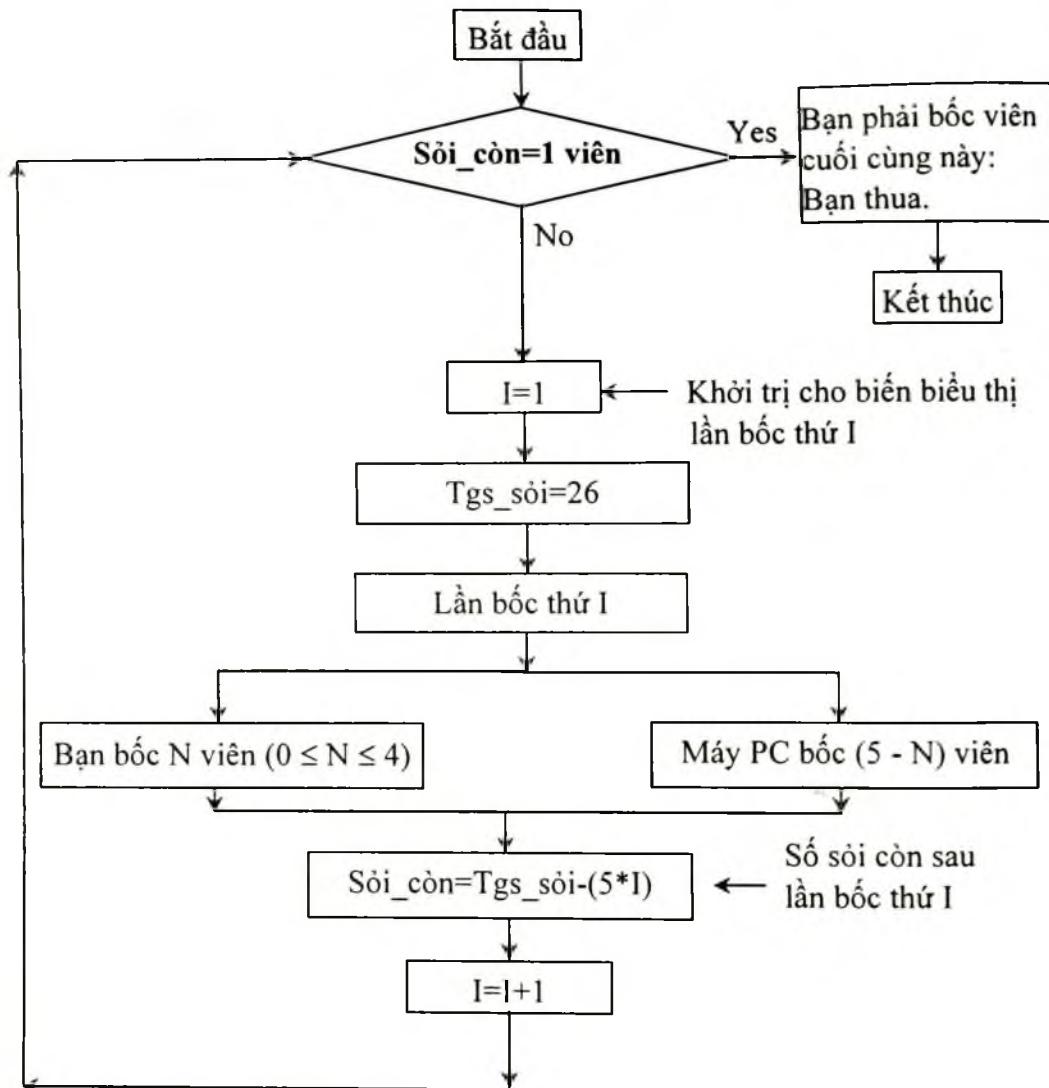
+ Bước  $S_i - (S \wedge S_i)$  viên ở đồng sỏi  $i$

```
int Way(int d, int v)      //Cách đi
{
    int S=Result();
    if (S==0)                  //Trường hợp thua
    {
        d=0;
        for(int i=1; i<=S.Len; ++i)
        {
            if(S[{i}] > S[d]) d=i;    //Trong đó
            S[d]=(Max(S[i]; i=0, 1.., N-1)

            if(S[d] > 0)
            {
                v=1;
                ++d;
            }
        }
        for(d=0; d< S.Len; ++d)
        if((s & S[d]) < s[d])
        {
            v=S[d]-(s & S[d]);
            ++d;
        }
    }
    return S;
}
```

## II. Diễn đạt thuật toán bằng Flowchart

### 1.11. Flowchart của bài toán bốc sỏi “sơ đẳng” (Bài 1.2)



## III. Diễn đạt thuật toán bằng ngôn ngữ lập trình bậc cao C/C++

### 1.23. (Bài 1.2: Trò chơi bốc sỏi sơ đẳng)

```

#include<iostream.h>
#include<conio.h>
main()
  
```

```

{ int i=1,n,tgsoi=26,soicon; clrscr();
  while (soicon!=1)
  { cout<<"\n Lan boc thu "<<i<<":";
    cout<<"\nBan boc may vien:"; cin>>n;
    cout<<"\n May tinh boc:"<<5-n;
    cout<<"\n Tong soi con lai sau lan boc thu
          "<<i<<": "<<tgsoi-(5*i);
    soicon=tgsoi-5*i;
    i=i+1;
  }
  cout<<"\n Ban phai boc vien cuoi cung. Ban thua
        roi!";
  getch();
}

```

1.24. (Trò chơi Nim) Xem phần code mô tả chiến thuật chơi ở mục 1.5 ∈ I ở trên.

1.25. Tập Nửa bội (Multi Half- Bài 1.6)

```

#include<stdio.h>
#include<conio.h>
#define ln 100
main()
{ int i,k2,k3,a2,a3,n,a[ln]; clrscr();
  printf("Nhập số phần tử:"); scanf("%d",&n);
  k2=k3=0; a[0]=1;
  printf("Tập nửa bội cần tìm:\n");
  printf("1");
  for(i=1; i<n; i++)
  { a2=2*a[k2]+1; a3=3*a[k3]+1;
    if (a2<=a3) {a[i]=a2; k2++; }
    if (a3<=a2) {a[i]=a3; k3++; }
    printf(" %4d",a[i]);
  }
}

```

```

    }
    getch();
}

```

### 1.26. (Tam giác ký tự Đôi xứng-Bài 1.7)

```

#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{ char *ch="abcdefghijklmnopqrstuvwxyz";
int i=0; clrscr();
while ((2*i)<strlen(ch))
{ for (int j=0; j<strlen(ch)-i; j++)
    cout<<" ";
    for (j=i; j<2*i; j++)
        cout<<ch[j];
    for (j=2*i; j>=i; j--)
        cout<<ch[j];
    cout<<"\n";
    i++;
}
getch();
}

```

## IV. Đánh giá độ phức tạp (ĐPT) của hàm

1.36. Hiển nhiên ĐPT của nhị thức  $x+1$  là  $O(x)$ . Còn rõ ràng  $x^2+1 \leq 2x^2$  với  $\forall x > 1$  nghĩa là:  $\log(x^2+1) \leq \log(2x^2) = \log 2 + \log x^2 = \log 2 + 2\log x \leq 3\log x$  với  $\forall x > 2$  → ĐPT của  $\log(x^2+1)$  là  $O(\log x)$ . Còn  $3x^2$  có ĐPT là  $O(x^2)$  và vì  $x\log x \leq x^2$  với  $\forall x > 1$  và theo định lý 2 trên đây nên ta có ĐPT của  $f(x)$  đã cho là  $O(\max(x\log x, x^2)) = O(x^2)$ .

1.40. a- $O(n^5)$ ; b- $O(n^3 \cdot n!)$ ; e-  $O(n^2(\log n)^2)$ ; f- $O(n^{2^n})$

1.41. Hiển nhiên tồn tại các hằng số  $C_1, C_2, k_1, k_2$  sao cho  $|f(x)| \leq C_1|g(x)|$  với  $\forall x > k_1$  và  $|g(x)| \leq C_2|h(x)|$  với  $\forall x > k_2$ . Do vậy nếu chọn  $x > \text{Max}(k_1, k_2)$  sẽ có  $|f(x)| \leq C_1|g(x)| \leq C_1C_2|h(x)| \rightarrow$  chứng tỏ ĐPT của  $f(x)$  là  $O(h(x))$ .

1.42. Không! Vì ta có ngay một phản ví dụ khi lấy  $f(x) = \frac{1}{x^2}$  và  $g(x) = \frac{1}{x}$ .

1.43. Gợi ý: Trước tiên cần chứng minh:  $\sum_{j=2}^n \frac{1}{j} < \int_1^n \frac{1}{x} dx$  bằng cách chứng minh tổng diện tích của hình chữ nhật có chiều cao là  $1/j$  và chiều rộng từ  $j-1$  đến  $j$  với  $j=2, 3, 4, \dots, n$  nhỏ hơn diện tích nằm dưới đường cong  $1/x$  với  $x=2, 3, \dots, n$ .

## V. Đánh giá ĐPT một vài thuật toán

1.46. ĐPT của Quick Sort. Để đánh giá ĐPT của Quick Sort Algorithm (QSA), trước tiên ta nêu lên các giả thiết: phần tử mốc (dùng để phân hoạch dãy đã cho) được chọn ngẫu nhiên (chứ không là phần tử giữa dãy đã cho) và thực thi thuật toán này nhờ kỹ thuật đệ quy. Hiển nhiên, khi dãy không có phần tử nào hoặc dãy có một phần tử thì độ phức tạp thời gian thực hiện  $T(0)$  và  $T(1)$  của QSA là hằng, nghĩa là đầu tiên ta có:

$$T(0)=T(1)=1 \quad (=const)$$

Thời gian thực thi QSA bằng thời gian gọi 2 lần đệ quy cộng với thời gian tuyến tính chi phí cho phân hoạch dãy (việc chọn phần tử mốc cũng có ĐPT là hằng). Khi đó hệ thức cơ sở của QSA là:

$T(N)=T(i)+T(N-i-1)+cN$  (V\_1),  $c$ : một hằng nào đó,  $N$ : số phần tử của dãy đã cho. Ở đây  $i$  là số phần tử của dãy con  $S_i$ , ta sẽ xem xét các tình huống sau:

a. *Tình huống tồi tệ nhất*: Tình huống này ứng với phần tử mốc (Pivot) là phần tử nhỏ nhất. Với  $i=0$  (và ta không để ý giá trị  $T(0)=1$  vì nó vô nghĩa với tính toán của ta) thì ta có hệ thức đệ quy sau:

$$T(N)=T(N-1)+cN, \quad N>1 \quad (\text{V}_2)$$

Ta dùng hệ thức (V\_1) và lặp lại nó, ta được:

$$T(N-1)=T(N-2)+c(N-1) \quad (\text{V}_3)$$

$$T(N-2)=T(N-3)+c(N-2) \quad (\text{V}_4)$$

$$T(N-3)=T(N-4)+c(N-2) \quad (V\_5)$$

-----

$$T(2)=T(1)+c(2) \quad (V\_6)$$

Cộng vế đối vế tất cả các đẳng thức trên ta được:

$$T(N)=T(1)+c \sum_{i=2}^N i = O(N^2) \quad (V\_7)$$

b. *Tình huống tốt nhất*: Trong tình huống này, phần tử mốc là điểm giữa của dãy và dễ đơn giản tính toán ta giả sử mỗi con dãy đã phân hoạch là một nửa đúng của dãy đã cho. Dẫu rằng giả thiết này có vẻ “thiếu khả thi” nhưng vì ta chỉ quan tâm đến Big\_O ( $O(n)$ ) nên nó có thể chấp nhận được trong lập luận của chúng ta):

Với giả thiết như vậy ta có:

$$T(N)=2T(N/2)+cN \quad (V\_8)$$

Chia cả hai vế ( $V\_8$ ) cho  $N$  ta được:

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c \quad (V\_9)$$

Ta tiếp tục “đệ quy” ( $V\_9$ ) sẽ được các hệ thức sau:

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} \quad (V\_10)$$

$$\frac{T(N/4)}{N/8} = \frac{T(N/8)}{N/8} \quad (V\_11)$$

-----

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c \quad (V\_12)$$

Cộng vế đối vế các đẳng thức từ ( $V\_8$ ) đến ( $V\_12$ ) ta được:

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N \quad (V\_13)$$

Nghĩa là:  $T(N)=cN\log N+N$

ở đây  $c$  và  $N$  đều là hằng nên ta có:

$$T(N)=O(N\log N)$$

c. *Tình huống trung bình:* Bao giờ đây cũng là tình huống phức tạp nhất và khó nhất! Để đánh giá ĐPT trong tình huống này, ta giả sử mỗi mảng con (kết quả của phân hoạch) có kích thước như nhau và vì thế có xác suất  $1/N$  ( $N$  kích thước dãy ban đầu). Sự giả định trên là hoàn toàn có giá trị thực sự với phần tử mốc của ta và với chiến lược phân hoạch của ta. Với giả định trên đây trung bình giá trị của  $T(i)$  và  $T(N-i-1)$  là:

$$(1/N) \sum_{j=0}^{N-1} T(j)$$

Trong tình huống này đẳng thức (V\_1) trở thành:

$$T(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} T(j) \right] + cN \quad (\text{V\_14})$$

Nhân cả 2 vế của (V\_14) với  $N$  ta được:

$$N \cdot T(N) = 2 \left[ \sum_{j=0}^{N-1} T(j) \right] + cN^2 \quad (\text{V\_15})$$

Tiếp tục “đệ quy” theo quy luật của công thức (V\_15) ta có:

$$(N-1)T(N-1) = 2 \left[ \sum_{j=0}^{N-1} T(j) \right] + c(N-1)^2 \quad (\text{V\_16})$$

Trừ (V\_15) vào (V\_16) ta được:

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c \quad (\text{V\_17})$$

Ước lược các số hạng đồng dạng ở công thức (V\_17) ta có:

$$NT(N) = (N+1)T(N-1) + 2cN \quad (\text{V\_18})$$

Chia cả 2 vế của (V\_18) cho  $N(N+1)$  ta được:

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \quad (\text{V\_19})$$

Dựa vào quy luật của (V\_19) ta tiếp tục “đệ quy” thì có lần lượt các đẳng thức sau:

$$\frac{T(n-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N} \quad (\text{V\_20})$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N} \quad (\text{V\_21})$$

-----

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \quad (\text{V\_22})$$

Cộng vế đối vế các đẳng thức từ (v\_17) đến (V\_22) ta được:

$$\frac{T(N)}{N+1} = \underbrace{\frac{T(1)}{2}}_{\text{const}} + 2c \sum_{i=3}^{N+1} \frac{1}{i}$$

Số hạng thứ hai ở vế phải có giá trị là  $\log_e(N+1) + \gamma - 3/2$  trong đó  $\gamma \approx 0.577$  là hằng số Euler. Vì vậy:

$$\frac{T(N)}{N+1} = O(\log N) \text{ suy ra } T(N) = O(N \log N) \quad (\text{V\_23})$$

Ta đã “đi hơi dài”, tóm lại ĐPT của Quick Sort trong tình huống trung bình là  $O(N \log N)$  với  $N$  là cỡ của dữ liệu vào.

1.47. Mặc dù bản thân cấu trúc Heap có tính tiện ích lớn. Tuy nhiên trên thực tế một thực thi tốt Quick Sort thường đánh bại Heap Sort. Một ứng dụng phổ biến của Heap là dùng nó mô phỏng hàng đợi ưu tiên khá thuận tiện.

Việc tính toán chính xác với giả thiết tổng quát ĐPT của Heap Sort khá khó và phức tạp. Cho đến nay nó vẫn là bài toán mở. Trong tài liệu dẫn [1] tác giả Knut đã xác định được ĐPT của HeapSortt sắp si  $2N \log N$  (Xem mục 2.12.9, trang 101)

## VI. Sử dụng một vài kiểu dữ liệu cơ bản có cấu trúc

1.48. (Điểm Yên ngựa - dùng mảng 2\_chiều)

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

int timmin(int a[20][20], int sh, int sc)
    //Thuật toán tìm phần tử Min trên hàng
```

```
{  
    int k, min=a[sh][0];  
    for(k=1;k<sc;k++) if(a[sh][k]<min) min = a[sh][k];  
    return min;  
}  
int timmax(int a[20][20], int sh, int sc)  
    //Thuật toán tìm phần tử Max trên cột  
{  
    int k, max=a[0][sc];  
    for(k=1;k<sh;k++) if (a[k][sc]>max) max = a[k][sc];  
    return max;  
}  
int main()  
{ clrscr();  
    int i,j,n,m,a[20][20],kt=0;  
    printf ("Nhập số hàng và số cột của ma trận:");  
    scanf ("%d%d", &m, &n);  
    for (i=0;i<m;i++)  
        for (j=0;j<n;j++)  
        {  
            printf ("A[%d][%d] = ",i,j);  
            scanf ("%d", &a[i][j]);  
        }  
    printf("\n\n");  
    for (i=0;i<m;i++)  
    {  
        for (j=0;j<n;j++)  
            if(a[i][j]==timmin(a,i,n)&&a[i][j]  
                ==timmax(a,m,j))  
    }
```

```

        printf ("Co mot diem yen ngua la p_tu
                 A[%d][%d]=%d\n", i, j, a[i][j]);
        kt++;
    }
}
if(kt==0)printf ("\n Khong co diem yen ngua nao");
getch();
return 0;
}

```

1.51. (Chương trình quản lý sinh viên trên File. Có vấn đáp cho mỗi thao tác! Dùng struct lồng nhau và FILE. Lưu ý cách truy cập vào từng thành phần của struct lồng nhau)

*/\* Bạn cần chú ý cách truy xuất File trong code này, đó là thao tác mà lần đầu học C các bạn sinh viên đều khá lúng túng thậm chí mất phương hướng: không biết dùng lệnh gì và với cú pháp nào để viết nó! \*/*

```

#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "string.h"
#include "ctype.h"
struct ngay
{
    int ngay;
    int thang;
    int nam;
};
struct quan_ly
{
    long int ma;
    char ht[50];

```

```
int gioi_tinh;
struct ngay_ngay_sinh;
char que_quan[50];
char lop_hoc[50];
float dtb;

};

void nhap_danh_sach()
{
    quan_ly h, p;
    float diem;
    long int ma, temp[50], i = 0, done;
    FILE *f;
    f=fopen("Inpu2.dat", "a+b");
    //Kiem tra trong file inpu2.dat co trung ma khong
    while(fread(&p,sizeof(quan_ly),1,f)>0)
    {
        temp[i++] = p.ma;
    }
    while(1)
    { printf("\nNhập mã số sinh viên: ");
        scanf("%ld%c", &ma);
        for(int j = 0; j < i; j++)
        {
            if(temp[j] == ma) done = 1;
        }
        if(done == 1)
        {
            done = 0;
            printf("Mã số này đã tồn tại! Yêu cầu\nnhập lại: ");
        }
    }
}
```

```
    }
    else break;
}

h.ma = ma;

nhap:printf("\nHo va ten sinh vien: ");
gets(h.ht);
if(h.ht[0]=='\0') goto nhap;
printf("\nNhap gioi tinh (Nu:0 Nam:1 ):");
scanf("%d%c",&h.gioi_tinh);

printf("\nNgay thang nam sinh (nam nhap
        4 so): ");
scanf("%d%d%d%c",&h.ngay_sinh.ngay,&h.ngay_sinh
        .thang,&h.ngay_sinh.nam);

printf("\nQue quan: "); gets(h.que_quan);
printf("\nSinh vien lop: "); gets(h.lop_hoc);
printf("\nDiem trung binh: ");
scanf("%f%c",&diem); h.dtb=diem;
fwrite(&h,sizeof(quan_ly),1,f);
fclose(f);

};

void xem_tat_ca_danh_sach()
{ clrscr();
int i=0;
quan_ly h;
FILE *f;
f=fopen("inpu2.dat","rb");
printf("\n DANH SACH TOAN BO SINH VIEN");
printf("\n");
printf("\nSTT | MA SO | HO VA TEN | GT | NAM
        SINH | QUE QUAN | LOP | DTB");
```

```
printf("\n-----");
while(fread(&h,sizeof(quan_ly),1,f)>0)
{
    i++;
    printf("\n%3d|",i);
    printf("%-7ld |",h.ma);
    printf("%-20s|",h.ht);
    printf("%2d |",h.gioi_tinh);

    printf("%2d/%2d/%4d|",h.ngay_sinh.ngay,h.ngay_sinh
        .thang,h.ngay_sinh.nam);
    printf("%-11s|",h.que_quan);
    printf("%-7s|",h.lop_hoc);
    printf("%2.2f",h.dtb);
    printf("\n");
}
fclose(f);
};

void tim_kiem(long &ma)
{ clrscr();
    quan_ly h;
    int i = 0;
    FILE *f;
    f=fopen("inpu2.dat","rb");
    printf("\n THONG TIN VE SINH VIEN CO MA SO:
        %ld",ma);
    printf("\n");
    while (fread(&h,sizeof(quan_ly),1,f)>0)
    { if(h.ma==ma)
```

```
{ printf("\nMa so sinh vien: %ld",ma);
    printf("\nHo va ten sinh vien: %s",h.ht);
    printf("\nGioi tinh: %d",h.gioi_tinh);
    printf("\nNgay thang nam sinh: %d/%d/%d",
           h.ngay_sinh.ngay,h.ngay_sinh.thang,
           h.ngay_sinh.nam);
    printf("\nQue quan: %s",h.que_quan);
    printf("\nSinh vien lop: %s",h.lop_hoc);
    printf("\nDiem trung binh: %2.2f",h.dtb);
    printf("\n");
    i = 1; break;
}
if(i == 0)
{ printf("\nKhong tim thay sinh vien co ma so
        %ld.",ma);
    printf("\n");
}
fclose(f);
};

void xoa_khoi_danh_sach(long &ma)
{ clrscr();
    int i=0,kt=0;
    char kiem_tra;
    quan_ly h, temp[50];
    FILE *f;
    f=fopen("inpu2.dat","r+b");
    printf("\n THONG TIN VE SINH VIEN MUON XOA:
        %ld",ma);
    printf("\n");
    while(fread(&h,sizeof(quan_ly),1,f)>0)
```

```
{ if (h.ma==ma)
    { printf("\nMa so sinh vien: %ld",ma);
      printf("\nHo va ten sinh vien: %s",h.ht);
      printf("\nGioi tinh: %d",h.gioi_tinh);
      printf("\nNgay thang nam sinh: %d/ %d/
%d",h.ngay_sinh.ngay,h.ngay_sinh.thang,
h.ngay_sinh.nam);
      printf("\nQue quan: %s",h.que_quan);
      printf("\nSinh vien lop: %s",h.lop_hoc);
      printf("\nDiem trung binh: %2.2f",h.dtb);
      printf("\n");
      kt=1;
    }
  else
  { temp[i] = h;
    i++;
  }
}
fclose(f);
if (kt==1)
{ printf("\n Co muon xoa hay khong? (c/k)");
  kiem_tra=getch();
  if(kiem_tra=='c'||kiem_tra=='C')
  {
    fopen("inpu2.dat", "wb");
    for(int j = 0; j < i; j++)
    {
      h = temp[j];
      fwrite(&h,sizeof(quan_ly),1,f);
    }
  }
}
```

```
printf("\nDa xoa xong sinh vien co ma so
      %ld.",ma);
fclose(f);
}
else
{ printf("\n");
  printf("\nViec xoa bi huy bo.");
}
}
else printf("\nKhong co sinh vien co ma so
      %ld.",ma);
};

void main()
{
char kiem_tra;
int i,j,done=0;
do
{ clrscr();
  printf("\n QUAN LY SINH VIEN ");
  printf("\n MENU CHINH ");
  printf("\n 1: Nhập dữ liệu cho các sinh viên.");
  printf("\n 2: Xem danh sách tất cả sinh viên.");
  printf("\n 3: Xóa lịch của một sinh viên.");
  printf("\n 4: Tìm kiếm và xem lý lịch của sinh
        viên theo mã.");
  printf("\n 5: Thoát.");
  printf("\nNhập số (1 đến 6) để lựa chọn nhu
        cầu: ");
  scanf("%d%c",&i);
  switch(i)
```

```
{case 1:  
    { tiep:  
        nhap_danh_sach();  
        printf("\nCo tiep tuc nhap hay khong (an k  
              de ve menu chinh) ?");  
        kiem_tra=getch();  
        if(kiem_tra=='c'||kiem_tra=='C') goto tiep;  
        else break;  
    }  
case 2:  
    { xem_tat_ca_danh_sach();  
        printf("\n ");  
        printf("\nCo quay lai menu chinh khong?  
              (c/k)");  
        char kiem_tra=(getch());  
        if(kiem_tra=='c'||kiem_tra=='C') done=0;  
        else done=1; break;  
    }  
case 3:  
    { clrscr();  
        xoa_tiep:  
        clrscr();  
        long int ma ;  
        printf("\nNhap ma can xoa: ");  
        scanf("%ld%c", &ma);  
        xoa_khoi_danh_sach(ma);  
        printf("\n");  
        printf("\nCo thuc su ban can xoa doi tuong  
              nay khong?(An k de khong xoa va tro  
              ve Menu!)?" );
```

```
    kiem_tra=getch();
    if(kiem_tra=='c'||kiem_tra=='C')
        goto xoa_tiep;
    else done=0;
    break;
}

case 4:
{ clrscr();
  tim_tiep:
  clrscr();
  long int ma ;
  printf("\nNhap ma can tim: ");
  scanf("%ld%c",&ma);
  tim_kiem(ma);
  printf("\n");
  printf("\nCo tiep tuc tim hay khong(an k
         de tro ve menu chinh) ?");
  kiem_tra=getch();
  if(kiem_tra=='c'||kiem_tra=='C')
      goto tim_tiep;
  else done=0; break;
}

default:
{
  done = 1;
  break;
}
}

} while (!done);
}
```

## Chương 2

# TÌM KIẾM TRONG VÀ SẮP XẾP TRONG

Chương 2 sẽ trình bày các thuật toán tìm kiếm và sắp xếp dữ liệu ở trong bộ nhớ RAM (ngầm định ta sẽ nói gọn là *tìm kiếm và sắp xếp*).

Trong tất cả các việc xử lý dữ liệu, việc tìm kiếm dữ liệu xảy ra thường ngày. Có thể nói không có việc gì không liên quan đến tìm kiếm dữ liệu hay thông tin.

Đến nay tồn tại hai thuật toán tìm kiếm cơ bản và nổi tiếng:

- Tìm kiếm tuần tự (Sequential Search- có người gọi là tìm kiếm tuyến tính).
- Tìm kiếm nhị phân.

Dưới đây, ta lần lượt xét các thuật toán đó.

## A. CÁC THUẬT TOÁN TÌM KIẾM

### 2.1. TÌM KIẾM TUẦN TỰ (SEQUENTIAL SEARCH)

Thuật toán này tìm kiếm một phần tử thuộc một tập hữu hạn tùy ý nào đó (tập các số nguyên, tập các ký tự, tập các từ, tập họ và tên các nhân viên. . .). *Phần tử cần tìm* gọi là *khóa tìm kiếm*. Để trực quan, ta xét tập các số nguyên và biểu diễn nó bằng mảng một chiều (gọi gọn là *dãy*).

#### 2.1.1. Ý tưởng của thuật toán

+ So sánh các phần tử thuộc dãy với khóa, bắt đầu từ phần tử ở đầu dãy. Có hai tình huống:

+ Nếu không tìm thấy khóa thì đánh dấu tình huống này bằng một trị số nào đấy (ví dụ -1).

+ Còn nếu tìm thấy khóa tại vị trí  $i$  nào đấy thì ghi nhớ vị trí  $i$  này. (nếu lập trình với hàm thì trả về vị trí  $i$  đó).

#### 2.1.2. Các bước của thuật toán

Bước 1:  $i=0$  //bắt đầu tìm ở vị trí đầu dãy; trong C/C<sup>++</sup> mặc định vị trí đầu dãy là 0

Bước 2: Chừng nào mà chưa duyệt hết dãy ( $i < n$ ) đồng thời chưa tìm thấy khóa  $x$  ( $a[i] \neq x$ ) thì:

- Bước 2.1. Tìm tiếp ở vị trí tiếp theo ( $i=i+1$ )
- Bước 2.2. Nếu đã duyệt tới cuối dãy mà không thấy khóa  $x$  thì đánh dấu tình huống này bằng  $-1$  và thoát khỏi công việc.
- Bước 2.3. Còn nếu tìm thấy khóa tại vị trí  $i$  nào đó ( $a[i]=x$ ) thì ghi nhớ vị trí  $i$  đó. Kết thúc việc tìm.

### 2.1.3. Cài đặt

*Hàm mô tả thuật toán tìm tuần tự*

```
int seq_find(int a[], int n, int x)
{
    int i=0;
    while ((i<n) && (a[i]!=x))
        i++;
    if (i==n) return -1;
    else return i;
}
```

**Chú ý:** Ở hàm này, mỗi lần phải làm 2 so sánh. Bạn có thể “cải tiến” nó để mỗi lần chỉ cần thực hiện 1 so sánh không?

*Code hoàn chỉnh dùng hàm (còn gọi là lập trình theo modul)*

```
#include<stdio.h>
#include<conio.h>

int seq_find(int a[], int n, int x) //Hàm diễn đạt thuật
                                    //toán tìm tuần tự
{
    int i=0;
    while ((i<n) && (a[i]!=x))
        i++;
    if (i==n) return -1; //Nếu đã duyệt hết dãy mà không tìm
                        //thấy khóa thì hàm trả về -1
    else return i; //còn nếu thấy khóa ở vị trí i thì hàm trả về giá trị i
}

main()
{
    int n,mang[50],i,x,tt; clrscr();
```

```

printf("\n Nhập số phần tử của mảng:");
scanf("%d", &n);
for(i=0; i<n; i++)
{ printf("a[%d]=", i);
scanf("%d", &mang[i]);
}
printf("\nNhập vào giá trị cần tìm:");
scanf("%d", &x);
tt=seq_find(mang, n, x);
if (tt>=0) printf("Tim thay %d tại vị trí thứ
%d", x, tt); //Biết tt >=0 đánh dấu kết quả tìm thấy
else printf("Không có %d trong dãy vừa nhập!", x);
//ngược lại không có khóa trong dãy
getch();
}

```

Cài đặt *không có hàm-lập trình phi modul*. (cài đặt này gần gũi với nhận thức của bạn hơn về thuật toán, chắc chắn là như vậy!)

```

#include<stdio.h>
#include<conio.h>
int a[100]; int n, i, x, ghinho, timthay=0;
main()
{ clrscr();
printf("\nNhập n="); scanf("%d", &n);
for (i=1; i<=n; i++)
{ printf("\n Nhập phần tử thứ %d:", i);
scanf("%d", &a[i]); }
printf("\n Nhập số cần tìm:"); scanf("%d", &x);
for (i=1; i<=n; i++)
if (a[i]==x)
{ timthay=1; ghinho=i; }

```

```

if (timthay) printf("Tim thay %d o vi tri thu
                     %d", x, ghinho);
else printf("Khong co %d trong day vua nhap.", x);
getch();
}

```

### 2.1.4. Độ phức tạp (ĐPT) của tìm kiếm tuần tự

Hiện nhiên độ phức tạp của thuật toán được đánh giá thông qua số lượng các phép so sánh thực hiện suốt chiều dài của dãy để tìm khóa x. Ta có các tình huống đánh giá như sau:

Tình huống	Số lần so sánh	Ý nghĩa
Tốt nhất	1	Tìm thấy khóa x ngay ở vị trí đầu tiên
Xấu nhất	n	Duyệt hết dãy mới trả lời được kết quả tìm
Trung bình	n/2	Duyệt đến giữa dãy thì tìm thấy khóa x

Tóm lại độ phức tạp của thuật toán tìm kiếm tuần tự là  $O(n)$  - tỷ lệ với cỡ của dữ liệu vào.

## 2.2. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH)

### 2.2.1. Những lưu ý đầu tiên về thuật toán này

Tìm kiếm nhị phân (Binary Search) áp dụng cho các dãy phần tử *đã có trật tự* (tăng hoặc giảm). Binary Search sử dụng một nguyên lý nổi tiếng của “Lý thuyết thông tin”(Information Theory): Khi xử lý bất cứ công việc gì, nếu ta biết cách làm giảm độ bất định đi một nửa thì tự khắc ta sẽ tăng được sự xác định (sự hiểu biết, còn gọi là lượng thông tin) lên gấp đôi. Trên cơ sở này ta sẽ xây dựng được ý tưởng của thuật toán:

### 2.2.2. Ý tưởng của thuật toán

Vì dãy đã có trật tự (ví dụ trật tự tăng) nên các phần tử trong dãy có quan hệ tuyến tính  $a_1 < a_2 < a_3 < \dots < a_i < a_{i+1}$ . Giả sử ta phải xác định xem khóa x có trong dãy không. Bất luận bài toán cụ thể nào cũng chỉ có 2 tình huống:

- +  $x < a_i$  thì chắc chắn x chỉ tồn tại trong đoạn  $[a_1, a_{i-1}]$ .
- +  $x > a_i$  thì chắc chắn x chỉ tồn tại trong đoạn  $[a_{i+1}, a_n]$

Trong cả 2 trường hợp, người ta gọi  $a_i$  là mốc để so sánh (có người gọi là chốt-Pivot-).

Để dùng luật “Giảm độ bất định lớn nhất” trong mọi công việc của “Lý thuyết thông tin” một cách hiệu quả nhất trong bài toán Binary Search, người ta thường *chọn phần tử mốc ở giữa hoặc gần giữa* dãy đã cho. Tại mỗi bước, ta so sánh khóa  $x$  với phần tử giữa dãy, tùy theo kết quả so sánh mà ta giới hạn việc tìm kiếm tiếp theo khóa  $x$  ở đoạn nào thuộc dãy đã cho: ở đoạn bên trái điểm giữa hay ở đoạn bên phải điểm giữa.

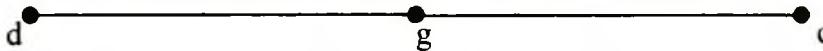
Như vậy trước tiên ta phải chia đôi dãy đã cho (nghĩa là tính điểm giữa của dãy  $g = (d+c)/2$ ; (/ là toán tử chia của C/C++)).

Để xác định ta giả sử khóa  $x < a[g]$ , tức  $x$  thuộc đoạn  $a[1] \rightarrow a[g-1]$ . Ta lại chia đôi đoạn ấy. Rồi lại xét xem  $x$  nằm bên trái hay phải điểm giữa của đoạn con đó. Khi đã có kết quả của việc này phải lặp lại việc tính điểm giữa của đoạn con và điểm đầu  $d$  (nếu  $x$  nằm bên phải mốc) hoặc điểm cuối  $c$  (nếu  $x$  nằm bên trái mốc). Xem minh họa hình 2.1 dưới đây. Tiếp theo lại chia đôi đoạn con mới xuất hiện mà khóa  $x$  nằm trong nó... Quy trình trên tiếp diễn cho tới khi đoạn con cuối cùng suy biến thành điểm giữa. Có 2 tình huống xảy ra:

- $x \equiv a[g]$  ( $x$  trùng với phần tử giữa)  $\rightarrow$  Tìm thấy khóa trong dãy.
- $x \neq a[g]$   $\rightarrow$  không có khóa trong dãy.

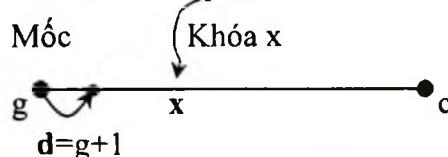
### *Sơ đồ minh họa cơ chế làm việc của Binary Sort*

- Dãy cho trước (để trực quan, biểu diễn bởi đoạn thẳng dc, d: đầu, c: cuối):



Hình 2.1. Minh họa cơ chế làm việc của Binary Sort ( $g$ : điểm giữa)

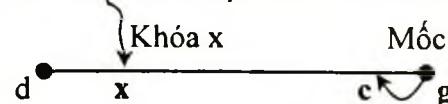
- Khóa cần tìm  $x > a[g]$  ( $a[g]$  là mốc để so sánh với  $x$ ,  $g$  là điểm giữa của dãy)  $\rightarrow$  tức  $x$  nằm trên đoạn con ở *bên phải* mốc:



Hình 2.1a. Khóa  $x$  thuộc (đâu đó) trên đoạn con bên phải điểm giữa  $g$

Trường hợp này, phải xác định điểm đầu của đoạn con trên đây:  $d=g+1$ ;

- Khóa  $x < a[g]$   $\rightarrow$  tức  $x$  nằm trên đoạn con *bên trái* mốc:



Hình 2.1b. Khóa  $x$  thuộc (đâu đó) trên đoạn con bên trái điểm giữa  $g$

Trường hợp này phải tính điểm cuối của đoạn con bên trái mốc như sau:  $c=g-1$

- Các thao tác trên đây sẽ lặp đi lặp lại cho tới khi đã xét hết các phần tử thuộc dãy và đoạn con phát sinh trở thành điểm duy nhất g. Khi đó kết thúc thuật toán.

### 2.2.3. Các bước của thuật toán

Bước 1: Gán  $d=1$ ;  $c=n$  ( $d$ : vị trí đầu tiên,  $c$ : vị trí cuối cùng của dãy)

Bước 2: Trong khi chưa xét hết các phần tử của dãy (tức là  $d \leq c$ ) thì lặp lại các việc sau:

Bước 3: Tính điểm giữa (chia đôi dãy)  $g=(d+c)/2$ .

Bước 4: (So sánh khóa với mốc là phần tử ở giữa dãy):

+ Bước 4.1: Nếu  $x < a[g]$  thì  $c=g-1$ ; //Tính điểm *cuối* của đoạn con hiện hành chứa khóa x

+ Bước 4.2: Còn nếu  $x > a[g]$  thì  $d=g+1$ ; //Tính điểm *đầu* của đoạn con hiện hành chứa khóa x

+ Bước 4.3: Còn  $a[g]=x$  (tìm thấy khóa x) → dừng.

+ Bước 4.4: Ngược lại ( $a[g] \neq x$ ) không có khóa trong dãy thì dừng.

### 2.2.4. Cài đặt

```

include <stdio.h>
#include<conio.h>
#define MAX 7

int tim_nhi_phan(int a[], int n, int x)
{ int g, d=0, c=n-1, k=-1;
  while (d <= c)
    { g=(d+c)/2;           //Xác định điểm giữa g
      if (x < a[g]) c=g-1; //Tìm khóa k ở dãy con bên trái
                             //điểm g và xác lập điểm cuối mới c
      else if(x>a[g]) d=g+1; //Ngược lại tìm khóa k ở dãy
                                //con bên phải điểm g và xác lập điểm đầu mới d
      else
    }
}

```

```

    { k=g; break; }      //Còn nếu tìm thấy khóa k
}
//thì hàm trả về khóa k ấy
return k;
}

void main()
{ int mang[MAX], i, x, k; clrscr();
printf("Nhập vào 7 phần tử với trait từ tăng dần
của dãy:\n");
for (i=0; i<MAX; i++)
{ printf("a[%d]=",i);
scanf("%d", &mang[i]);
}
printf("\nNhập vào giá trị cần tìm:");
scanf("%d", &x);
k=tim_nhi_phan(mang,MAX,x);
if (k>=0)
    printf("\nGiá trị %d tìm thấy tại vị trí thứ
%d",x,k);
else
printf("\nKhông tìm thấy %d trong dãy!",x);
getch();
}

```

### 2.2.5. Độ phức tạp của Binary Search

Tình huống	Số lần so sánh	Ý nghĩa
Tốt nhất	1	Phần tử giữa dãy có giá trị bằng khóa x
Xấu nhất	$\log_2 n$	Duyệt hết dãy mới tìm thấy x hoặc không thấy x
Trung bình	$\log_2 (n/2)$	Giả sử mọi phần tử trong dãy có xác suất như nhau

Từ bảng trên, độ phức tạp của thuật toán Binary Sort là  $O(\log_2 n) < \text{ĐPT}$  của thuật toán tìm tuần tự =  $O(n)$  → với cùng cỡ dữ liệu vào n thì Binary Sort chạy nhanh hơn Sequential Search nhiều.

## B. CÁC THUẬT TOÁN SẮP XẾP

### 2.3. MỞ ĐẦU

Bên cạnh việc tìm kiếm dữ liệu là việc sắp xếp dữ liệu cũng không kém quan trọng. Bởi muốn tra cứu nhanh thì phải biết cách tổ chức, sắp xếp dữ liệu khoa học. Bạn hãy tưởng tượng, một từ điển Việt-Anh có 500 ngàn từ (từ điển nhỏ) để lộn xộn, không xếp theo trật tự từ điển thì liệu bạn có thể tra cứu nhanh một từ bất kỳ không? Tất nhiên câu trả lời của bạn là không! Một ví dụ giản dị như vậy cho ta thấy giữa sắp xếp dữ liệu và tra cứu nó luôn có quan hệ mật thiết không tách rời được.

### 2.4. LIỆT KÊ CÁC THUẬT TOÁN SẮP XẾP

Trong mục này ta sẽ xét đầy đủ, chi tiết về lý thuyết cũng như cài đặt một số thuật toán sắp xếp dữ liệu.

#### 2.4.1. Các thuật toán sắp xếp đơn giản, thông dụng

Các thuật toán này gồm có:

- + Sắp xếp lựa chọn (Selection Sort)
- + Sắp xếp đổi chỗ (Interchange Sort)
- + Sắp xếp chèn (Insertion Sort)
- + Sắp xếp nổi bọt (Bubble Sort)

#### 2.4.2. Các thuật toán khó (khi cài đặt) và ít quen thuộc đối với học sinh, sinh viên

- + Sắp xếp rung-lắc (Shake Sort)
- + Sắp xếp với độ dài bước giảm dần (Shell Sort)
- + Sắp xếp nhanh (Quick Sort)
- + Sắp xếp vun đống (Heap Sort)
- + Sắp xếp trộn (Merge sort)
- + Sắp xếp bùu điện (Post Sort, tên khác: Radix Sort=sắp xếp cơ số)

## 2.5. SẮP XẾP LỰA CHỌN (SELECTION SORT)

### 2.5.1. Ý tưởng của thuật toán

Tìm phần tử nhỏ nhất (ký hiệu là min) trong dãy đã cho có độ dài n, rồi đặt nó lên đầu dãy. Sau đó “lờ” nó đi, xét dãy hiện hành có n-1 phần tử và tìm phần tử nhỏ nhất của dãy này rồi đặt nó lên đầu dãy. Lặp lại các thao tác trên với dãy hiện hành cho đến khi dãy đang xét chỉ còn một phần tử: đến đây dãy hình thành sau n-1 bước đã xếp tăng (phần tử duy nhất trong dãy hiện hành cuối cùng nói trên là phần tử lớn nhất).

Để tìm được phần tử min, nhất thiết ta phải so sánh phần tử min với các phần tử  $a[i]$  thuộc dãy ( $i=2, 3, \dots, n$ ). Rồi đổi chỗ hai phần tử  $a[i]$  và min. Việc đổi chỗ hai phần tử này dựa vào khái niệm **nghịch thế**. Việc đổi chỗ hai phần tử trên nhằm mục đích hủy nghịch thế.

### 2.5.2. Nghịch thế

Cho trước dãy các số  $a_1, a_2, \dots, a_n$ . Gọi  $i, j$  là các chỉ số của hai phần tử  $a_i, a_j$ . Nếu có  $i < j$  và  $a_i > a_j$  thì ta gọi đó là **một nghịch thế**.

### 2.5.3. Sơ đồ mô tả cơ chế hoạt động Selection Sort (Còn gọi là sơ đồ “dò vết”)

Cho dãy số: 12 5 -1 32 20 -9 56

0	1	2	3	4	5	6	← Index
12	5	-1	32	20	-9	56	

$i=1 \rightarrow ptmin=-9$ , đặt nó lên đầu dãy (vị trí 0): tức đổi chỗ  $a[0]=12$  và  $a[5]=-9$

0	1	2	3	4	5	6	← Index
-9	5	-1	32	20	12	56	

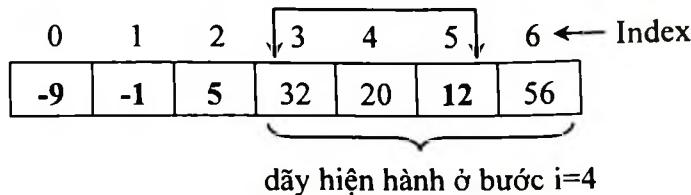
dãy hiện hành ở bước  $i=2$

$i=2 \rightarrow$  ở dãy hiện hành,  $ptmin=-1$ , đặt -1 lên đầu dãy này ta có:

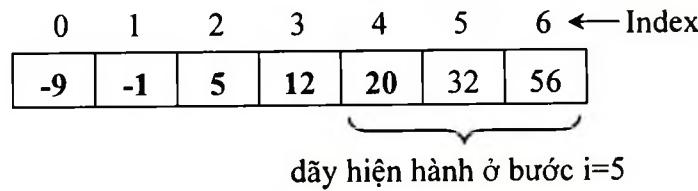
0	1	2	3	4	5	6	← Index
-9	-1	5	32	20	12	56	

dãy hiện hành ở bước  $i=3$

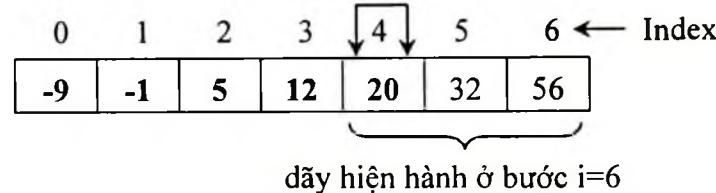
$i=3 \rightarrow ptmin=5$ , nó đang ở vị trí đầu của dãy hiện hành, đổi chỗ “tại chỗ” cho nó ta có:



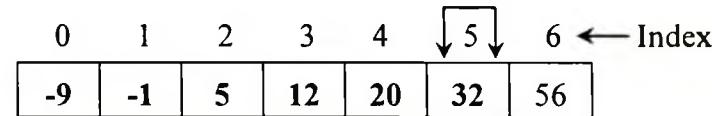
$i=4 \rightarrow ptmin=12$ , đặt nó lên đầu dãy hiện hành ta có:



$i=5 \rightarrow ptmin=20$ , nó đang ở vị trí đầu dãy, đổi chỗ “tại chỗ” cho nó ta được:



$i=6 \rightarrow ptmin=32$ , nó đang ở phía đầu dãy (một chỗ đã hợp lệ), đổi chỗ “tại chỗ” cho nó ta được:



#### 2.5.4. Các bước thực thi Selection Sort

Để nhanh ta duyệt từ hai đầu dãy lại.

Bước 1:  $i=0$ ; Trong khi  $i \leq n$  (chưa duyệt hết dãy đã cho) làm:

+ Bước 1.1:  $vitrimin=i$ ; //khởi trị cho vị trí tồn tại phần tử min.

+ Bước 1.2:  $j=i+1$ ; Trong khi  $j \leq n$  (tức chưa duyệt hết dãy đã cho):

- Bước 1.2.1: Nếu  $a[j] < a[vitrimin]$  thì

```
{
    vitrimin=j;
}
```

- **Bước 1.2.2:** Gọi hàm Doicho(a[j], a[vitrimin]) để đổi chỗ 2 phần tử a[j] và a[vitrimin]

**Bước 2:** Nếu  $i < n-1$  thì  $i = i+1$ ; Lặp lại Bước 1.2. Ngược lại dừng.

Nếu  $j \leq i$  thì  $j = j+1$ ; Lặp lại Bước 1.2.1. Ngược lại dừng.

### 2.5.5. Hàm diễn đạt thuật toán Selection Sort

```
void SelectionSort(int a[], int n)
{ int i, j, vitrimin;
for(i=0; i<n-1; i++)
{
    vitrimin=i;
    for(j=i+1; j<n; j++)
        if(a[j]<a[vitrimin]) //Phát hiện nghịch thế
            vitrimin=j;           //Ghi nhớ vị trí tồn tại nghịch thế
    Doicho(a[i], a[vitrimin]); //Đổi chỗ để hủy nghịch thế, tạo
                                //trật tự cục bộ
}
}
```

### 2.5.6. Cài đặt Selection Sort

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
void Nhap(int a[], int n)
{ int i;
for(i=0; i<n; i++)
{ printf("Nhập a[%d]:", i);
scanf("%d", &a[i]);
}
}
void Xuat(int a[], int n)
{ int i;
for(i=0; i<n; i++)
```

```
    printf("%4d", a[i]);
}

void Doicho(int &a, int &b)
{ int tg;
  tg=a; a=b; b=tg;
}

//Selection Sort Algorithm
void SelectionSort(int a[], int n)
{ int i, j, vitrimin;
  for(i=0; i<n-1; i++)
  { vitrimin=i;
    for(j=i+1; j<n; j++)
      if(a[j]<a[vitrimin])          //tìm nghịch thế
        vitrimin=j;                  //ghi nhớ vị trí có giá trị min
    Doicho(a[i], a[vitrimin]); //hủy nghịch thế
  }
}

void main()
{ int a[MAX], n; clrscr();
  printf("Nhập số phần tử mảng n=");
  scanf("%d", &n);
  Nhập(a, n);
  printf("Mảng vừa nhập:\n");
  Xuất(a, n);
  SelectionSort(a, n); printf("\n\n");
  printf("Mảng sau khi sắp xếp bằng phương pháp
        Lựa Chọn :\n");
  Xuất(a, n);
  getch();
}
```

### 2.5.7. Độ phức tạp của Selection Sort

Từ sơ đồ mô tả cơ chế hoạt động và mục các bước của thuật toán, rõ ràng ta thấy ở mỗi bước i luôn cần  $n-i$  so sánh để tìm phần tử nhỏ nhất ở dãy hiện hành. Biểu thức tính  $n-i$  không phụ thuộc vào tình trạng của dãy đã cho, do đó trong mọi trường hợp ta có ĐPT của thuật toán là:

$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

Mỗi một hoán vị dùng 3 phép gán. Số phép gán lại phụ thuộc vào trạng thái ban đầu của dãy đã cho. Do đó ta chỉ có thể ước lược trong 2 tình huống sau:

Tình huống	Số lần so sánh	Số phép gán
Tốt nhất	$n(n-1)/2$	0
Xấu nhất	$n(n-1)/2$	$3n(n-1)/2$

## 2.6. SẮP XẾP CHÈN (INSERT SORT)

### 2.6.1. Những nhận xét và quan niệm ban đầu

Bất luận tình trạng cụ thể của dãy đã cho ra sao, ta cũng luôn nhìn thấy một đoạn con của dãy này đã có trật tự (tăng hoặc giảm).

Dãy hay một đoạn con (của dãy đang xét) chỉ gồm 1 phần tử duy nhất xem như *đã có trật tự*.

### 2.6.2. Ý tưởng của thuật toán

Cho trước dãy  $a_1, a_2, \dots, a_n$ . Thế nào cũng có i phần tử đầu tiên  $a_1, a_2, \dots, a_{i-1}$  *đã có trật tự* (từ nhận xét đầu trên đây).

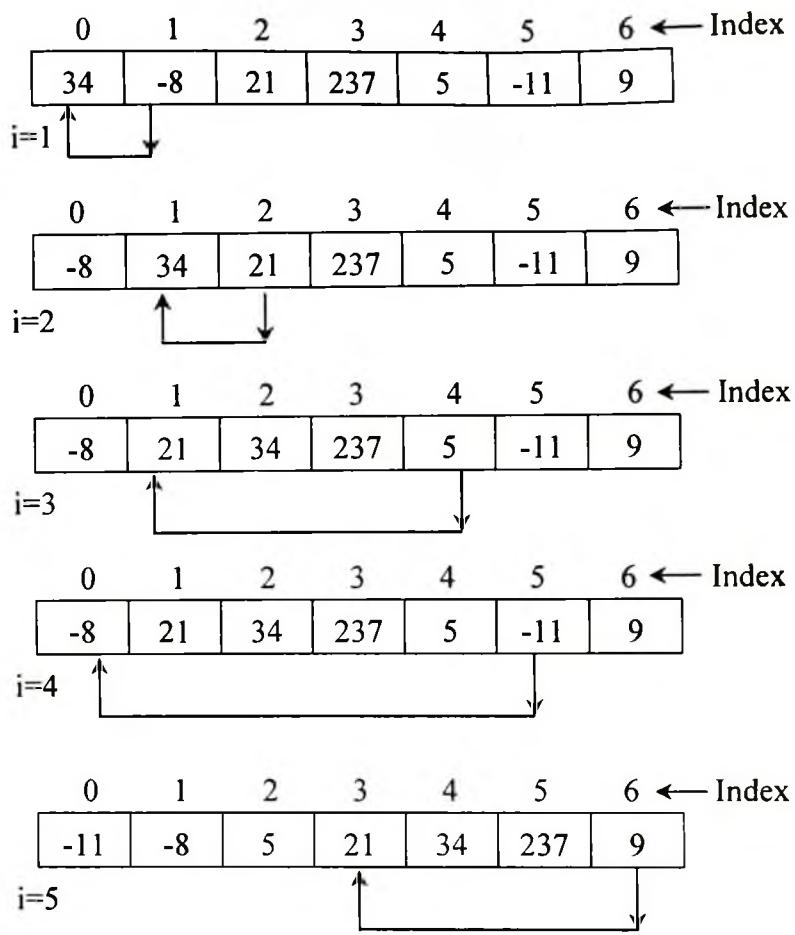
Vấn đề đặt ra là phải tìm vị trí thích hợp để chèn  $a_i$  vào đoạn con trên sao cho dãy kết quả vẫn bảo tồn trật tự của đoạn con hiện hành.

Vị trí này chính là vị trí ở giữa 2 phần tử  $a_{k-1}$  và  $a_k$  (với  $1 \leq k \leq i$ ) mà  $a_{k-1} \leq a_i \leq a_k$ .

Dãy đã cho  $a_1, a_2, \dots, a_n$  có đoạn con chỉ gồm 1 phần tử  $a_1$  *đã có trật tự* (quan niệm đã nêu trên!). Bây giờ ta chỉ việc thêm  $a_2$  vào đoạn con  $a_1$  ta được  $a_1a_2$  *đã được sắp*; tiếp theo thêm  $a_3$  vào đoạn  $a_1a_2$  được đoạn  $a_1a_2a_3$  *được sắp*. Tiếp tục cho đến khi chèn thêm  $a_n$  vào đoạn  $a_1, a_2, \dots, a_{n-1}$  ta được dãy  $a_1, a_2, \dots, a_n$  *có trật tự* (tăng hoặc giảm).

### 2.6.3. Sơ đồ mô tả cơ chế làm việc của Insert Sort (sơ đồ dò vết)

Cho dãy số: 34 -8 21 237 5 -11 9



Kết thúc: 0 1 2 3 4 5 6 ← Index

-11	-8	5	21	34	237	9
-----	----	---	----	----	-----	---

### 2.6.4. Các bước thực thi Insert Sort

Ta dùng hai vòng lặp để thực thi thuật toán này.

**Vòng lặp ngoài:** Duyệt từ đầu dãy ( $i=1$ ) đến cuối dãy ( $i=n$ ) để tìm vị trí thích hợp chèn phần tử  $x$  vào mà vẫn bảo toàn trật tự của đoạn con hiện hành.

Bước 1: Khởi trị vtchen=i-1; //khi i=1, vtchen=0 có nghĩa là coi đoạn chỉ có 1 // phần tử a[0] đã được sắp. (vtchen: vị trí để chèn)

Bước 2: Khởi trị cho phần tử cần chèn là x=a[i]

Vòng lặp trong: Thực hiện lặp các việc sau khi vtchen  $\geq 0$  và phần tử cần chèn  $x < a[\text{vtchen}]$ :

Bước 3: Đẩy các phần tử sang phải một vị trí để tìm chỗ thích hợp chèn x vào đoạn con đã có trật tự (việc này tương đương với phép gán:

$a[\text{vtchen}+1]=a[\text{vtchen}]$ )

Bước 4: Sau mỗi lần thực hiện Bước 3, giảm vtchen đi một ( $\text{vtchen}=\text{vtchen}-1$ )

Bước 5: Chèn x vào vị trí thích hợp đã tìm được ( $a[\text{vtchen}+1]=x$ , đó là vị trí thỏa mãn điều kiện ở đầu vòng lặp trong)

Bước 6: Vòng lặp trong kết thúc khi vtchen vượt ra ngoài phạm vi dãy và  $x=a[\text{vtchen}]$ .

*Vòng lặp ngoài kết thúc khi  $i=n$ .*

### 2.6.5. Hàm diễn đạt thuật toán Insert Sort

```
void direc_inse_sort(int a[], int n)
{
    int i, vtchen, x;
    for(i=1; i<n; i++)
    {
        x=a[i];
        vtchen=i-1;
        while (vtchen>=0 && a[vtchen]>x)
            {a[vtchen+1]=a[vtchen];
             vtchen--;}
        a[vtchen+1]=x;
    }
}
```

### 2.6.6. Cài đặt Insert Sort

```
#include "stdio.h"
#include "conio.h"
```

```
#define MAX 100

void input (int a[],int n)
{ int i;
for(i=0;i<n;i++)
{ printf("Nhập a[%d]:",i); scanf("%d",&a[i]); }

void out(int a[],int n)
{ int i;
for(i=0;i<n;i++)
printf("%4d",a[i]);

}

void insertSort(int a[],int n)      //Thuật toán xép chèn
{ int vtchen,i,x;
for(i=1;i<n;i++)
{ int x=a[i];
vtchen=i-1;
while((vtchen>=0)&&(x<a[vtchen]))
{ a[vtchen+1]=a[vtchen];
vtchen--;
}
a[vtchen+1]=x;
}
}

void main()
{ int a[MAX],n; clrscr();
printf("Nhập số phần tử n="); scanf("%d",&n);
input(a,n);
printf("Dãy vừa nhập:");
out(a,n); printf("\n\n");
```

```

InsertSort(a, n);
printf("Day sau khi sap xep bang th_toan Chen
       truc tiep:\n");
out(a, n);
getch();
}

```

### 2.6.7. Độ phức tạp của Insert Sort

Ở thuật toán này các phép so sánh quan trọng thực hiện trong mỗi lần lặp while xác định vtchen còn chưa nhỏ hơn 0 và (quan trọng hơn) tìm vị trí thích hợp, và mỗi lần tìm thấy vị trí không thích hợp thì dịch chuyển (đẩy) phần tử ở vị trí không thích hợp ấy ( $a[vtchen]$ ) sang phải một vị trí. Thuật toán thực hiện  $n-1$  vòng lặp while và số lượng so sánh và rời chỗ phụ thuộc vào trạng thái ban đầu của dãy đã cho nên ta có bảng đánh giá ĐPT sau:

Tình huống	Số lần so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{n-1} i = n-1$	$\sum_{i=1}^{n-1} 2i = 2(n-1)$
Xấu nhất	$\sum_{i=1}^{n-1} (i-1) = \frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (i+1) = \frac{n(n+1)}{2} - 1$

## 2.7. SẮP XẾP ĐỔI CHỖ (INTERCHANGE SORT)

### 2.7.1. Mở đầu

Nhiều thuật toán sắp xếp thường thực hiện đổi chỗ hai phần tử kề liền nhau trên suốt chiều dài của dãy ban đầu để thu được dãy có trật tự (tăng hoặc giảm). Thực chất của việc đổi chỗ hai phần tử là *khử nghịch thế*.

### 2.7.2. Ý tưởng

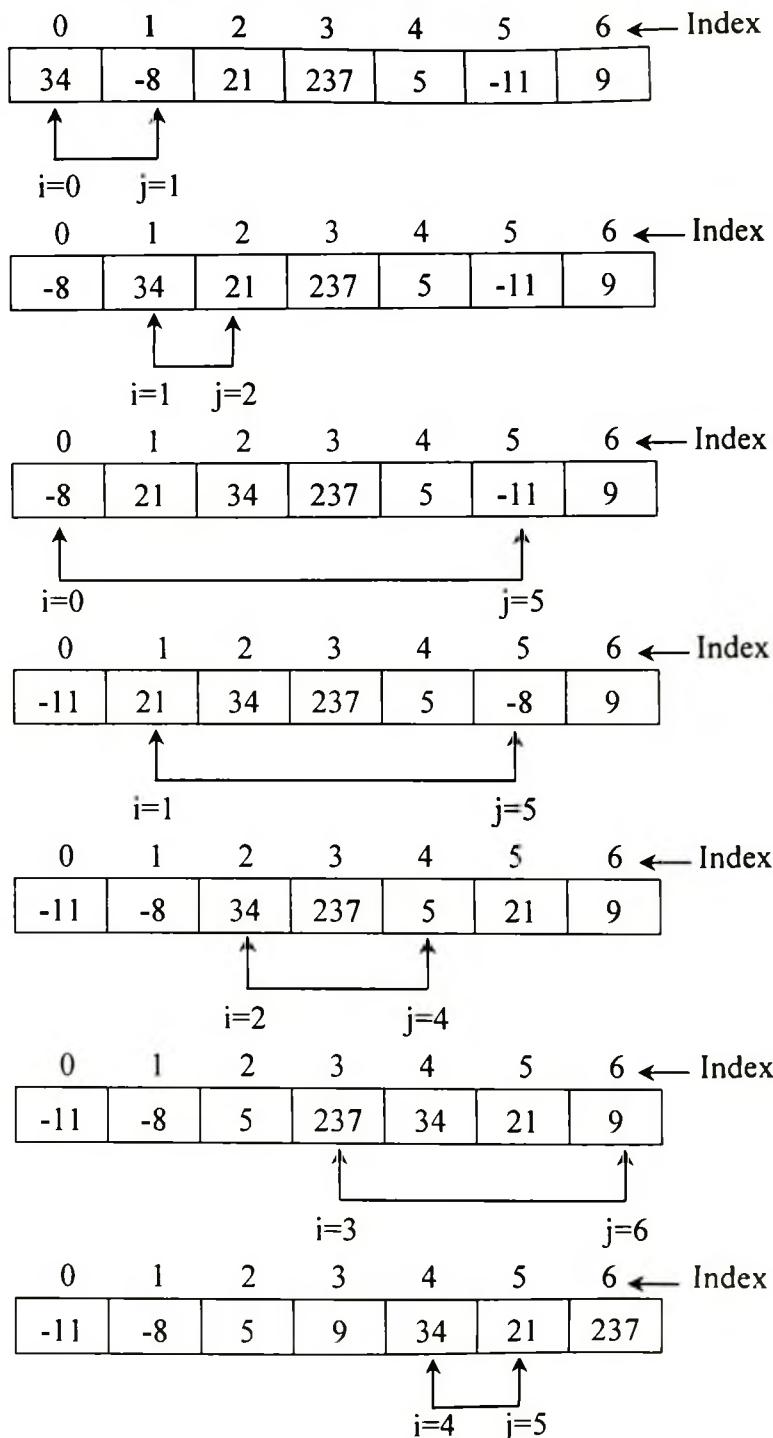
Dãy đã sắp (đã có trật tự) thì không có nghịch thế. Chẳng hạn ta xét dãy đã sắp theo trật tự tăng. Trường hợp này,  $a_0$  là phần tử nhỏ nhất,  $a_1$  là phần tử nhỏ thứ hai, v.v..., cụ thể tình trạng của dãy ấy là:  $a_0 \leq a_1 \leq a_2 \leq \dots \leq a_n$ .

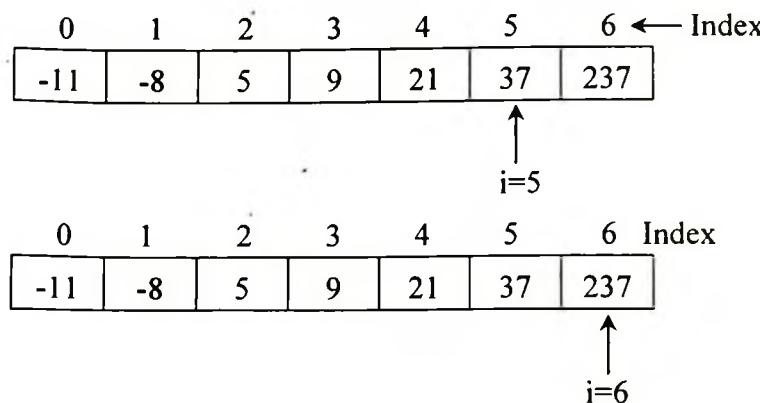
Một dãy tùy ý cho trước (đang lộn xộn) luôn tồn tại nghịch thế, bao giờ cũng xuất phát từ phần tử đầu dãy hiện hành rồi tìm tất cả các cặp nghịch thế chứa phần tử ấy.

Và để sắp dãy có trật tự thì cần khử các nghịch thế đó bằng cách *đổi chỗ* cặp nghịch thế này.

### 2.7.3. Sơ đồ mô tả cơ chế hoạt động của Interchange Sort (sơ đồ dò vết)

Cho dãy số: 34 -8 21 237 5 -11 9





#### 2.7.4. Các bước thực thi Interchange sort

- Bước 1:  $i=0$ ; // Xét từ vị trí (phần tử) đầu tiên thuộc dãy
- Bước 2:  $j=i+1$ ; //Bắt đầu tìm các nghịch thế trong dãy và đổi chỗ chúng
- Bước 3: Trong khi  $i \leq n$  thì làm:
  - + Bước 3.1: Nếu  $i < j$  mà  $a[i] > a[j]$  thì gọi hàm Doicho( $a[i]$ ,  $a[j]$ ) để đổi chỗ 2 phần tử  $a[i]$ ,  $a[j]$
  - + Bước 3.2:  $j=j+1$ ;
- Bước 4:  $i=i+1$ ;
- Bước 4.1: Nếu  $i < n$  lặp lại B2;
- Bước 4.2: Ngược lại: Dừng;

#### 2.7.5. Hàm diễn đạt thuật toán Interchange Sort

```
void Xepdoicho(int a[], int n)
{
    int i, j;
    for (i=0; i<n-1; i++)
        for (j=i+1; j<n; j++)
            if (a[j]<a[i])
                Doicho(a[i], a[j]);
}
```

#### 2.7.6. Độ phức tạp của Interchange Sort

Trong thuật toán này số các phép đổi chỗ phụ thuộc vào kết quả so sánh. Ta có bảng đánh giá độ phức tạp của thuật toán dưới đây:

Tình huống	Số lần so sánh	Số lần đổi chỗ
Tốt nhất	$\sum_{i=1}^{n-1} (n-i+1) = \frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (n-i+1) = \frac{n(n-1)}{2}$

## 2.8. SẮP XẾP NỐI BỌT (BUBBLE SORT)

### 2.8.1. Xuất xứ của cái tên “Nối bọt” (Bubble)

Thuật ngữ này được các nhà Toán-Tin thế giới sử dụng xuất phát từ một nhận xét thực tế: khi ta khuấy một cốc coffee (hoặc cốc trà) thì các phần tử nhẹ nhất trong cốc này là các bọt khí sẽ nổi lên trên cùng. Thuật toán sắp xếp này thông thường cũng *đặt các giá trị nhỏ nhất (“nhẹ nhất”) lên đầu dãy*.

### 2.8.2. Ý tưởng của thuật toán

- + Duyệt từ đầu (hoặc cuối) dãy tìm các cặp nghịch thế và đổi chỗ chúng, đưa phần tử nhỏ hơn (hoặc lớn hơn) về đầu dãy hiện hành (hoặc cuối) nếu xếp tăng dần (nếu xếp giảm dần).

- + Ở bước xử lý tiếp theo không xét đến phần tử đó nữa và lặp lại thao tác đầu cho tới khi đã xét hết mọi cặp phần tử trong dãy.

### 2.8.3. Các bước thực thi Bubble sort

Bước 1:  $i=0;$  //Bắt đầu duyệt xuôi từ đầu dãy

Bước 2:  $j=n-1;$  //Mở vòng duyệt trong: duyệt ngược từ cuối dãy

- + Bước 2.1: Trong khi  $j > i$  thực hiện các thao tác:

Nếu  $a[j] < a[j-1]$  thì  $Doicho(a[j], a[j-1]);$

- + Bước 2.2:  $j=j-1;$

Bước 3:  $i=i+1;$  //Duyệt xuôi ở bước kế tiếp

- + Bước 3.1: Nếu  $i < n-1$  quay lại bước 2

- + Bước 3.2: Ngược lại dừng;

### 2.8.4. Sơ đồ dò vết của Bubble Sort (tương tự các trường hợp trên, độc giả tự làm)

### 2.8.5. Hàm diễn đạt thuật toán Bubble Sort

```
void bubb_sort(int a[], int m)
{ int i, j, tg;
  for(i=0; i<n-1; i++)
    for(j=n-1; j>i; j--)
      if (a[j]<a[j-1])
        {tg=a[j]; a[j]=a[j-1]; a[j-1]=tg; }
}
```

### 2.8.6. Độ phức tạp của thuật toán

Tình huống	Số lần so sánh	Số lần đổi chỗ
Tốt nhất	$\sum_{i=1}^{n-1} (n-i+1) = \frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (n-i+1) = \frac{n(n-1)}{2}$

## 2.9. SẮP XẾP RUNG (SHAKE SORT)

### 2.9.1. Ý tưởng của thuật toán

Shake sort là cải tiến của Bubble Sort theo mục tiêu: “không những phần tử nhẹ nhất “nổi” lên trên đầu dãy mà cả phần tử nặng nhất cũng “chìm” xuống cuối dãy”. Sự việc này giống như ta rung lắc một cốc nước giải khát! Như vậy Shake sort phải điều khiển cả hai quá trình này một cách tự nhiên và tự động. Muốn thế ta phải ghi nhớ lần đổi chỗ cuối cùng khi duyệt ngược dãy từ cuối lên và khi duyệt xuôi dãy từ đầu xuống cuối để quyết định ở bước tiếp theo sẽ duyệt từ đó (chỗ ghi nhớ) đến đâu.

### 2.9.2. Các bước thực thi Shake Sort

Bước 1: //Khởi trị

- left=0; right=n-1; k=n-1; //k: vị trí trung gian ghi nhớ các lần đổi chỗ

Bước 2: Trong khi mà left<right thì lặp lại các việc sau:

+ Bước 2.1: Duyệt ngược từ cuối dãy lên đầu dãy làm:

- Bước 2.1.1: Nếu a[j]<a[j-1] thì:

Bước 2.1.1.1: Gọi hàm Hoanvi đổi chỗ  $a[j]$  và  $a[j-1]$ ;

Bước 2.1.1.2: Ghi nhớ vị trí  $j$  vào biến  $k$ ;

- Bước 2.1.2: Ghi nhớ  $k$  vào biến  $left$  (tức ở bước tiếp, đầu dãy hiện hành là  $k$ );

+ Bước 2.2: Duyệt xuôi từ vị trí  $left$  (có được ở bước 2.1.2) làm:

- Bước 2.2.1: Nếu  $a[j] > a[j+1]$  thì:

Bước 2.2.1.1: Gọi hàm Hoanvi đổi chỗ  $a[j]$  và  $a[j+1]$ ;

Bước 2.2.1.2: Lưu  $j$  vào  $k$ ;

- Bước 2.2.2: Ghi nhớ  $k$  vào  $right$  (tức ở bước tiếp, cuối dãy là  $k$ );

Bước 3: Vòng lặp ngoài cùng kết thúc khi  $left > right$

### 2.9.3. Sơ đồ dò vết (độc giả tự làm)

### 2.9.4. Hàm diễn đạt thuật toán Shake Sort

```
void ShakerSort(int a[], int n)
{ int i, j;
    int left, right, k;
    left=0;
    right = n-1;
    k = n-1;
    while (left < right)
        { for (j = right; j > left; j --)
            if (a[j] < a[j-1])
                {
                    Hoanvi(a[j], a[j-1]);
                    k = j;
                }
            left = k;
        for (j = left; j < right; j++)
            if (a[j] > a[j+1])
                { Hoanvi(a[j], a[j+1]);
```

```

    k = j;
}
right = k;
}
}

```

### 2.9.5. Độ phức tạp của thuật toán Shake Sort

Tình huống	Số lần so sánh	Số lần đổi chỗ
Tốt nhất	$n \log_2 n$	$(n \log_2 n)/2$
Xấu nhất	$n^2$	$n^2$

## 2.10. SẮP XẾP VỚI ĐỘ DÀI BƯỚC GIẢM DẦN - CÒN GỌI LÀ SẮP XẾP THEO CÁCH NÃ PHÁO (SHELL SORT)

### 2.10.1. Ý tưởng của thuật toán Shell Sort

+ Shell Sort có thể xem là cải tiến của thuật toán chèn. Trong thuật toán này người ta chia dãy ban đầu *thành các dãy con có độ dài h* (trong Insert sort thì  $h=1$ ). Tiếp theo thực hiện sắp xếp các dãy con đó (Bằng một trong các thuật toán đã nêu trên). Giảm dần  $h$  ở mỗi bước tiếp theo cho tới khi  $h=1$  thì dừng.

+ Như vậy việc mấu chốt ở đây *quyết định tốc độ của thuật toán là chọn độ dài h ở mỗi bước và số bước sắp xếp*. Ví dụ ta quyết định  $k$  bước sắp xếp, khi ấy  $h$  cần chọn thỏa mãn các điều kiện sau:

$$h_i > h_{i+k}; h_k = 1;$$

+ Đến nay vẫn chưa có lý luận tổng quát để chọn  $h$ . Trong tài liệu tham khảo [5] của Knut, ông ta đề xuất cách chọn  $h$  và  $k$  cho một vài bài toán cụ thể như sau:

Chọn  $h_i = (h_{i-1}-1)/3$ ;  $h_k = 1$  và  $k = \log_3 n - 1$  cho dãy 127 40 13 4 1 và chọn  $h_i = (h_{i-1}-1)/2$ ;  $h_k = 1$  và  $k = \log_2 n - 1$  cho dãy 16 7 3 1

### 2.10.2. Các bước thực thi Shell Sort

Bước 1: Chọn công thức tính  $k$

Bước 2: Khởi trị cho  $step = 0$  (số bước sắp xếp)

Bước 3: Khởi trị cho độ dài= $h[step]$  và  $i=$ độ dài

Bước 4: (Sắp xếp các dãy con có độ dài= $h[step]$ )

- + Bước 3.1: Xác định công thức tính h (và cất kết quả tính vào mảng h);
- + Bước 3.2: Sắp dãy con dài h theo thuật toán lựa chọn;
- + Bước 3.3: Nếu step < k: Quay lại bước 3.2;
- + Bước 3.4: Ngược lại (tức step ≥ k): Dừng sắp xếp các đoạn con;
- Bước 5: i=i+1;
- + Bước 5.1: Nếu  $i \leq n$  quay lại Bước 4; //n-độ dài dãy ban đầu
- + Bước 5.2: Ngược lại: Kết thúc;

### **2.10.3. Sơ đồ dò vết (độc giả tự làm)**

### **2.10.4. Hàm diễn đạt thuật toán Shell Sort**

```

void ShellSort(int a[], int n, int h[], int k)
{
    int step, i, j, x, dodai;
    for (step=0 ; step <k; step++)
    {
        dodai=h[step];
        for (i=dodai; i <n; i++)
        {
            x=a[i]; j=i-dodai;
            while ((x<a[j]) && (j>=0)) // xép dãy con chứa x
                nhờ thuật toán lựa chọn
                {a[j+dodai]=a[j];
                 j=j-dodai;
                }
            a[j+dodai]=x;
        }
    }                                //Kết thúc for theo i
}                                    //Kết thúc for theo step
}

```

### **2.10.5. Cài đặt Shell Sort**

```

#include "stdio.h"
#include "conio.h"
#include "math.h"
#define MAX 100

```

```
void Nhap(int a[], int n)
{ int i;
    for(i=0; i<n; i++)
        {printf("Nhập a[%d]:", i); scanf("%d", &a[i]);}
}
void Xuat(int a[], int n)
{ int i;
    for(i=0; i<n; i++)
        printf("%4d", a[i]);
}
void Hoanvi(int &a, int &b)
{ int tg; tg=a; a=b; b=tg; }
void Taoday(int n, int h[], int &k) //Tạo dãy lưu độ dài h
                                         tính bởi công thức h[i]=2*h[i+1]+1
{ k=(int)(log(n)/log(2));           //Tính k=logn/log2
    int i;
    h[k]=1;
    for(i=k-1; i>=0; i--)
        h[i]=h[i+1]*2+1;
    k++;           //vị trí cho phần tử sau cùng trong dãy là k
}
void ShellSort(int a[], int n, int h[], int k)
{ int step, i, j, x, dodai;
    for (step=0 ; step <k; step++)
        { dodai=h[step];
            for (i=dodai; i <n; i++)
                { x=a[i]; j=i-dodai;
                    while ((x<a[j]) && (j>=0)) // Xếp dãy con chứa x
                        bằng phương pháp Select Sort
                        a[j]=x;
                    j--;
                }
        }
}
```

```

        {a[j+dodai]=a[j]; //sort
        j=j-dodai;
    }
    a[j+dodai]=x;
}
}

void main()
{ int a[MAX],n,h[MAX],k; clrscr();
printf("Nhập số phần tử mảng n=");scanf("%d",&n);
Nhập(a,n);
printf("Đây vừa nhập là:\n");
Xuất(a,n);
printf("\n");
Taoday(n,h,k);
ShellSort(a,n,h,k);
printf("Đây sau khi xếp nhỏ thuận toán Shell
Sort:\n");
Xuất(a,n);
getch();
}

```

### **2.10.6. Độ phức tạp của thuật toán Shell sort**

Đến nay chưa có một đánh giá tổng quát cho Shell Sort algorithm. Chỉ có “nghiệm cục bộ” của vấn đề này cho một vài trường hợp cụ thể. Nếu chọn độ dài  $h$  theo đề xuất của Knut [5]  $h[i]=3.h[i+1]+1$  và  $i=\text{round}(\log_3 n)-1$  thì ta có đánh giá sau:

- Số lần so sánh:  $1.66n^{1.22}$
- Số lần đổi chỗ:  $1.66n^{1.22}$

## **2.11. SẮP XẾP NHANH (QUICK SORT)**

### **2.11.1. Ý tưởng của thuật toán**

+ Để đạt được mục đích của thuật toán, người ta phân chia dãy ban đầu thành 3 dãy con.

+ Muốn vậy người ta chọn một phần tử làm mốc. Về lý luận phần tử này có thể lấy bất cứ phần tử nào trong dãy đã cho (lấy chử không phải chọn!). Thông thường để xác định và do đó thuận tiện cho phân tích thuật toán, người ta chọn mốc là điểm giữa của dãy ban đầu:

$$m = (l+r)/2 \quad // l: vị trí đầu dãy, r: vị trí cuối dãy$$

+ Sau khi đã có mốc  $m$ , người ta chia dãy ban đầu thành 3 dãy con như sau:

- Dãy thứ nhất gồm các phần tử  $a_1, a_2, \dots, a_i$  không lớn hơn  $a_m$ , tức là:

$$a_k \leq a_m \text{ với } \forall k=1,2,\dots,i$$

- Dãy thứ hai có phần tử  $a_k = a_m$  với  $k=i, \dots, j$

- Dãy thứ ba gồm các phần tử  $a_{i+1}, a_{i+2}, \dots, a_n$  không nhỏ hơn  $a_m$ , cụ thể:

$$a_k > a_m \text{ với } \forall k=j, \dots, n$$

$a_k < a_m$	$a_k = a_m$	$a_k > a_m$
-------------	-------------	-------------

Hình 2.1. Phân chia dãy ban đầu thành 3 dãy con

### 2.11.2. Các bước thực thi Quick Sort

Bước 1: (Khởi trị)  $i=l$ ;  $j=r$ ; và xác định mốc  $x=mang[(l+r)/2]$ ;

Bước 2: (Phân chia các dãy con)

+ Bước 2.1: Trong khi mà phần tử  $i$  của dãy  $< x$  thì tạo dãy con thứ nhất:

$$a_1 \dots a_i < x; i++;$$

+ Bước 2.2: Trong khi mà phần tử  $j$  của dãy  $> x$  thì tạo dãy con thứ 2:

$$a_{j+1} \dots a_{r-1} > x; j--;$$

+ Bước 2.3: (Hủy nghịch thế) Nếu  $i \leq j$  // Mà  $mang[i] \geq x \geq mang[j]$  thì  
Đổi chỗ( $mang[i], mang[j]$ );

Bước 3: (Sắp xếp các dãy con nhờ đệ quy) Trong khi mà  $i \leq j$  thì lặp lại  
các thao tác:

+ Bước 3.1: Nếu  $i < j$ : sắp dãy con từ  $i$  đến  $j$ ;

+ Bước 3.2: Nếu  $i < r$ : sắp dãy con từ  $i$  đến  $r$ ;

Bước 4: + Nếu  $i < j$  (chưa xét hết dãy) thì quay lại bước 2;

+ Ngược lại  $i \geq j$ : dừng;

*Chú ý quan trọng:* Một số tác giả chọn phần tử mốc là phần tử đầu của dãy con hoặc phần tử cuối cùng. Hai cách này cho ra kết quả sắp xếp như nhau và không khác biệt đáng kể về ĐPT.

### 2.11.3. Bảng dò vết

Chẳng hạn ta cần xếp dãy  $a=77\ 44\ 99\ 66\ 33\ 55\ 88\ 22\ 44$ , ta có bảng dưới đây:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
22							77	
		33		99				
			44					66
	33	44						
				55	99			
					66			99
						77	88	

## Dãy dã xếp tăng

#### 2.11.4. Hàm diễn đạt thuật toán Quick Sort

```

void sort(int l, int r)
{ int i, j, x, w;
  i=l; j=r;
  x=mang[(l+r)/2];
  do {
    while (mang[i]<x)
      i++;
    while (mang[j]>x)
      j--;
    if (i<=j)
      {w=mang[i];
       mang[i]=mang[j];
       mang[j]=w;
      }
  } while (i<j);
}

```

Đổi chỗ các các  
cặp nghịch thế

```
        i++;
        j--;
    }
} while (i<=j);
if (l<j) sort(l,j);
if (i<r) sort(i,r);
}
```

### 2.11.5. Cài đặt Quick Sort

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 15
int mang[MAX];
void in_mang(int *mang)
{ int i;
    for (i=0; i<MAX; i++)
        printf("%4d", mang[i]);
}
void quick_sort(int l, int r)
{ int i, j, x, w;
    i=l;
    j=r;
    x=mang[(l+r)/2];
    do { while (mang[i]<x)
        i++;
        while (mang[j]>x)
            j--;
        if (i<=j)
            { w=mang[i];
```

```

        mang[i]=mang[j];
        mang[j]=w;
        i++;
        j--;
    }
} while (i<=j);
if (l<j) sort(l,j);
if (i<r) sort(i,r);
}
void quicksort()
{ sort(0, MAX-1); }
void main()
{ int i; clrscr();
    randomize();
    for (i=0; i<MAX; i++)
        mang[i]=random(15);
    printf("Day tu dong phat sinh, no chua duoc
           sap:\n");
    in_mang(mang); printf("\n\n");
    quicksort();
    printf("Day da xep tang nho thuật toán Quick
           Sort:\n");
    in_mang(mang);
    getch();
}
}

```

### 2.11.6. Độ phức tạp của thuật toán

Độ phức tạp của thuật toán này phụ thuộc mạnh vào việc chọn mốc. Trường hợp tốt nhất là cứ mỗi lần phân chia đều chọn mốc là giữa các dãy phát sinh. Trong tình huống này chỉ cần  $\log_2 n$  lần phân chia thì sắp xếp xong. Nếu chọn mốc không hợp lý (chọn phần tử đầu hoặc cuối dãy) thì phải cần  $n$  lần phân chia mới sắp xếp xong.

Tính huống	Số lần so sánh	Số lần đổi chỗ
Tốt nhất	$n \log 2n$	$n \log 2n / 2$
Xấu nhất	$n^2$	$n^2$

Trên thực tế Quick Sort chạy nhanh hơn Heap Sort đến 1/3 thời gian.

## 2.12. SẮP XẾP VÙN ĐÓNG (HEAP SORT)

### 2.12.1. Định nghĩa Heap

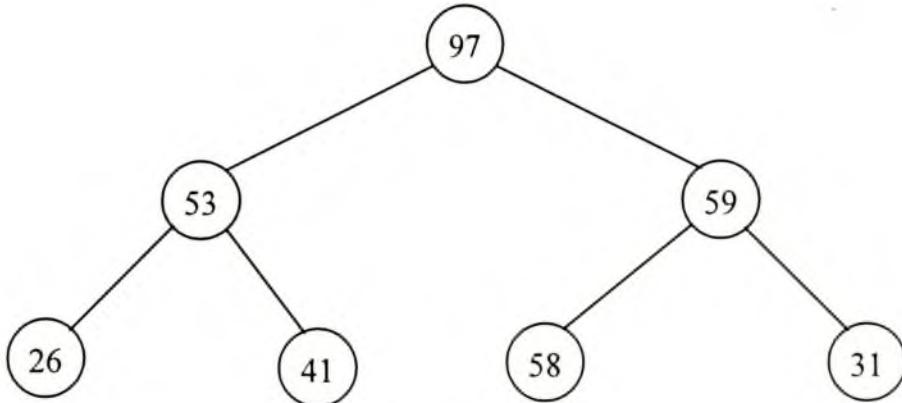
Dãy các phần tử  $a_1, a_2, \dots, a_r$  là Heap nếu với  $\forall i \in [l, r]$  thỏa các điều kiện dưới đây:

$$a_i \geq a_{2i}$$

$$a_i \geq a_{2i+1}$$

Các cặp  $(a_i, a_{2i})$  và  $(a_i, a_{2i+1})$  gọi là các cặp liên đới.

Ví dụ 2.1. Dãy 97 53 59 26 41 58 31 là một Heap. Để dễ hình dung, hãy xem sơ đồ biểu diễn Heap dưới đây (ở chương Cây (Tree) ta sẽ bàn thêm về Heap):



Hình 2.2. Sơ đồ (cây) biểu diễn Heap

### 2.12.2. Các tính chất của Heap

*Tính chất 1:* Nếu dãy  $a_1, a_2, \dots, a_r$  là Heap thì khi xóa bỏ một số phần tử ở hai đầu Heap, dãy còn lại vẫn là một Heap. Ở Ví dụ 2.1, sau khi xóa 2 phần tử ở đầu và cuối Heap thì dãy 59 26 41 vẫn là một Heap.

*Tính chất 2:* Nếu  $a_1, a_2, \dots, a_r$  là Heap thì phần tử  $a_1$  (đầu Heap) luôn là phần tử lớn nhất trong Heap.

### 2.12.3. Các bước thực thi Heap Sort

Thực thi thuật toán Heap Sort gồm 2 pha:

*Pha 1: Hiệu chỉnh dãy các phần tử đã cho thành Heap. (Khi cài đặt ta sẽ viết một modul làm việc này và một modul tạo Heap)*

*Pha 2: Gồm 2 bước:*

Bước 1: Đưa phần tử min về cuối dãy tức là Doicho( $a_1, a_r$ ); //  $r=n$

Bước 2:

+ Bước 2.1: Hủy phần tử min ra khỏi Heap, khi đó  $r=n-1$ ;

+ Bước 2.2: Hiệu chỉnh phần còn lại của dãy thành Heap;

Bước 3: Nếu  $r>1$  (chưa xử lý hết dãy đã cho)  $\rightarrow$  Quay lại bước 2. Còn thì ( $r=1$ ): Dừng.

### 2.12.4. Xây dựng hàm hiệu chỉnh dãy ban đầu thành Heap

```
void shift(int a[], int L, int r) //Hàm hiệu chỉnh dãy
{ int x, i, j;
  i=L;
  j=2*i+1;
  x=a[i];
  while(j<=r)
  { if(j<r)
    { if(a[j]<a[j+1])           //Tìm phần tử lớn nhất trong 2 phần tử
      j++;                      // a[j] và a[j+1]
    if(a[j]<=x) return; //Xét phần tử kế tiếp
    else
    { a[i]=a[j];
      a[j]=x;
      i=j;
      j=2*i+1;
      x=a[i];
    }
  }
}
```

### 2.12.5. Xây dựng hàm tạo Heap

```
void CreateHeap(int a[], int n) //Hàm tạo Heap
{
    int L;
    L=n/2-1;
    while(L>=0)
    {
        shift(a,L,n-1);
        L=L-1;
    }
}
```

### 2.12.6. Hàm thực thi Heap Sort

```
void HeapSort(int a[], int n) // Hàm thực thi thuật toán Heapsort
{
    int r;
    CreateHeap(a, n);
    r=n-1;
    while(r>0)
    {
        Doicho(a[0], a[r]); //a[0] là root của cây Heap
        r--;
        if(r>0)
            shift(a, 0, r);
    }
}
```

### 2.12.7. Cài đặt Heap Sort

```
#include "iostream.h"
#include "iomanip.h"
#include "stdio.h"
#include "conio.h"
void input(int a[], int n)
{
    int i;
    for(i=0; i<n; i++)
}
```

```

    { cout<<"a["<<i<<"]="; cin>>a[i]; }
}

void out(int a[],int n)
{ int i;
  for(i=0;i<n;i++)
    cout<<setw(4)<<a[i];
}

void Doicho(int &x,int &y)
{ int tg;
  tg=x; x=y; y=tg;
}

void shift(int a[],int L,int r) //Hàm hiệu chỉnh dãy
                                a[l] -> a[r] thành Heap
{ int x,i,j;
  i=L;
  j=2*i+1;
  x=a[i];
  while(j<=r)
  { if(j<r)
      if(a[j]<a[j+1])                         //Tìm phần tử lớn nhất trong
                                                    2 phần tử a[j] và a[j+1]
      j++;                                     //Xét phần tử kế tiếp
      if(a[j]<=x) return;
      else
      { a[i]=a[j];
        a[j]=x;
        i=j;
        j=2*i+1;
        x=a[i];
      }
  }
}

```

```
}

void CreateHeap(int a[],int n) //Hàm tạo Heap
{ int L;
L=n/2-1;
while(L>=0)
{ shift(a,L,n-1);
L=L-1;
}
}

void HeapSort(int a[],int n) //Thuật toán Heapsort
{ int r;
CreateHeap(a,n);
r=n-1;
while(r>0)
{ Doicho(a[0],a[r]);           //a[0] là phần tử đầu của Heap
r--;
if(r>0)
shift(a,0,r);
}
}

void main()
{ int a[20],n; clrscr();
cout<<"Nhập số phần tử n của dãy:";
cin>>n;
cout<<"Nhập giá trị cho từng phần tử của
dãy:"<<endl;
input(a,n);
cout<<"Dãy vừa nhập:"<<endl;
out(a,n); cout<<endl<<endl;
```

```

HeapSort(a, n);

cout<<"Day da xep tang bang th_toan VUN DONG
      (HEAP SORT) :"<<endl;

out(a, n);

getch();
}

```

### 2.12.8. Bảng đồ vật Heap Sort

Giả sử ta cần xếp dãy  $a=77\ 44\ 99\ 66\ 33\ 55\ 88\ 22\ 44$

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
	66		33					
99		88				77		
44								99
88		77	44			44		
22							88	
77		55			22			
44						77		
66	44							
22				66				
55		22						
33				55				
44			33					
33				44				
44	33							
22		44						
33	22							
22	33							

Dãy đã xếp tăng

### 2.12.9. Độ phức tạp của Heap sort

Việc đánh giá độ phức tạp của thuật toán Heap sort rất phức tạp. Theo Knut [5] ta có các đánh giá sau:

Tình huống	Số lần so sánh	Số lần đổi chỗ
Tốt nhất	$n \log 2n$	$n \log 2n - 0.9n$
Xấu nhất	$7n + 2n \log 2n$	$17n/2 + n \log 2n$

Tóm lại trong cả 2 tình huống, độ phức tạp của thuật toán này đều tỷ lệ với  $n \log 2n$ .

## 2.13. SẮP XẾP TRỘN (MERGE SORT)

### 2.13.1. Ý tưởng

+ Mọi dãy  $a_1, a_2, \dots, a_n$  tùy ý đều có thể xem là tập của *các dãy con kề nhau đã có thứ tự tăng*. Các dãy con ấy gọi là các Run. Dãy *chỉ gồm 1 phần tử cũng xem là dãy có trật tự tăng*. Ví dụ dãy 77 44 99 66 33 55 88 22 44 là tập của các Run  $\{77\}, \{44, 99\}, \{66\}, \{33, 55, 88\}, \{22, 44\}$

+ Với dãy đã cho ta tìm cách chia nó thành các Run sao cho Run có độ dài lớn hơn có số phần tử là lũy thừa cơ số 2.

Ví dụ dãy đã cho có thể chia thành các Run:  $\{77\}, \{44, 99\}, \{22, 33, 55, 88\}, \{44, 66\}$

+ Tiếp theo tìm cách phân phối các Run rồi trộn từng cặp Run với nhau để được Run có thứ tự tăng với độ dài lớn hơn. Việc này kết thúc khi chỉ còn một Run.

### 2.13.2. Các bước thực thi Merge Sort

Bước 1: (Khởi trị cho bước trộn k)  $k=1$ ;

Bước 2: Phân phối dãy đã cho  $a_1, a_2, \dots, a_n$  thành từng Run gồm k phần tử cho các dãy con b, c;

Bước 3: Trộn từng cặp Run  $\in$  dãy b, c rồi cất lên a.

Bước 4:

+ Bước 4.1:  $k=2*k$ ;

+ Bước 4.2: Nếu  $k < n$  quay lại bước 2;

Còn thì: Dừng;

### 2.13.3. Bảng dò vết

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
44	77							
		66	99					
	66	77						
						22	44	88
				22	33	44	55	
22	33	44	44	55	66	77	88	99

### 2.13.4. Xây dựng hàm phân phối các Run

```

void Phanphoi(int a[], int N, int &nb, int &nc,
int k)      //Phân phối các phần tử ở dãy ban đầu a cho các dãy con b và c
{
    int i, ja, jb, jc; //i, ja: chỉ số dãy đã cho; jb, jc: chỉ số các dãy con
    ja=jb=jc=0;
    while (ja<N)
        {
            for(i=0; (ja<N) && (i<k); i++, ja++, jb++)
                b[jb]=a[ja];
            for(i=0; (ja<N) && (i<k); i++, ja++, jc++)
                c[jc]=a[ja];
        }
    nb=jb;      //Cập nhật độ dài cho dãy con b
    nc=jc;      //Cập nhật độ dài cho dãy con c
}

```

### 2.13.5. Xây dựng hàm trộn các Run

```

void Tron(int a[], int nb, int nc, int k)//Thực thi thuật
                                            toán trộn
{
    int j, jb, jc, ib, ic, kb, kc; //j: chỉ số dãy đã cho;
                                    jb, jc, ib, ic: chỉ số các dãy con b, c.
    j=jb=jc=0; ib=ic=0;
}

```

```
while( (nb>0) && (nc>0) )      //nb, nc: độ dài các dãy con b, c
{
    kb=min(k,nb);  kc=min(k,nc);
    if(b[jb+ib]<=c[jc+ic])
    {
        a[j++]=b[jb+ib]; ib++;
        if(ib==kb)
        {
            for(;ic<kc;ic++)
                a[j++]=c[jc+ic];
            jb+=kb;
            jc+=kc;
            ib = ic=0;
            nb-=kb;
            nc-=kc;
        }
    }
    else
    {
        a[j++]=c[jc+ic]; ic++;
        if(ic==kc)
        {
            for(;ib<kb;ib++)
                a[j++]=b[jb+ib];
            jb+=kb;
            jc+=kc;
            ib = ic=0;
            nb-=kb;
            nc-=kc;
        }
    }
}
}
```

### 2.13.6. Xây dựng hàm xếp trộn

*Gồm 2 lời gọi hàm: Phân phối và hàm Trộn*

```
void XepTron(int a[], int N)
{ int k; nc=0; nb=0;
  for (k = 1; k < N; k*= 2)
  { Phanphoi(a, N, nb, nc, k);
    Tron(a, nb, nc, k);
  }
}
```

### 2.13.7. Cài đặt Merge Sort

```
#include "iostream.h"
#include "iomanip.h"
#include "conio.h"
#include "stdio.h"
#define MAX 100
int b[MAX], c[MAX], nb, nc;
//Phân phối các phần tử ở dãy ban đầu cho các dãy con b và c
void Phanphoi(int a[], int N, int &nb, int &nc,
               int k)
{ int i, ja, jb, jc;      //i, ja: chỉ số dãy đã cho; jb, jc: chỉ số
                           //các dãy con
  ja=jb=jc=0;
  while (ja<N)
  { for (i=0; (ja<N) && (i<k); i++, ja++, jb++)
    b[jb]=a[ja];          //Phân phối các Run từ dãy a lên dãy b
    for (i=0; (ja<N) && (i<k); i++, ja++, jc++)
    c[jc]=a[ja];          //Phân phối các Run từ dãy a lên dãy c
  }
  nb=jb;                  //Cập nhật các độ dài nb
  nc=jc;                  //và nc sau mỗi lần phân phối
```

```

}

int min(int a,int b) //Hàm tìm min trong 2 số cho trước để
                     sau này khi trộn
{ if(a>b) return b; //sẽ khử nghịch thế (nếu có)
  else return a;
}

void Tron(int a[],int nb, int nc,int k)//Thực thi thuật
                                         toán trộn
{ int j, jb, jc, ib, ic, kb, kc; //j: chỉ số dãy đã cho;
                                         jb, jc, ib, ib: chỉ số các dãy con b, c.
  j=jb=jc=0; ib=ic=0;
  while((nb>0)&&(nc>0))
  { kb=min(k,nb); kc=min(k,nc); //Cách chọn độ dài bước
                                         trộn kb và kc
    if(b[jb+ib]<=c[jc+ic]) //Nếu điều kiện này thỏa mãn thì
    { a[j++]=b[jb+ib]; ib++; //Lưu b[jb+ib] vào dãy a ở vị
                                         trí j; cập nhật ib, tăng j lên một để trộn tiếp
      if(ib==kb)
      { for(ic<kc; ic++)
        a[j++]=c[jc+ic];
        jb+=kb;
        jc+=kc;
        ib = ic=0;
        nb-=kb;
        nc-=kc;
      }
    }
    else
    { a[j++]=c[jc+ic]; ic++;
      if(ic==kc)
      { for(; ib<kb; ib++)
        }
    }
  }
}

```

```
a[j++]=b[jb+ib];
jb+=kb;
jc+=kc;
ib = ic=0;
nb-=kb;
nc-=kc;
}
}
}
}
void XepTron(int a[], int N)
{ int k;nc=0;nb=0;
for (k = 1; k < N; k *= 2)
{ Phanphoi(a, N, nb, nc, k);
Tron(a, nb, nc, k);
}
}
void Nhaph(int a[],int n)
{ int i;
for(i=0;i<n;i++)
{ cout<<"Nhaph a["<<i<<"]:";
cin>>a[i];
}
}
void Xuat(int a[],int n)
{ int i;
for(i=0;i<n;i++)
cout<<setw(4)<<a[i];
}
void main()
{ int a[MAX],n; clrscr();
cout<<"Nhaph so phan tu n cua day:";
```

```

    cin>>n;
    Nhap(a,n);
    cout<<"Day vua nhap la:"<<endl;
    Xuat(a,n);
    cout<<endl<<endl;
    XepTron(a,n);
    cout<<"Ket qua xep Tron day vua nhap:";
    Xuat(a,n);
    getch();
}

```

### 2.13.8. Độ phức tạp của Merge Sort

Trước hết ta thấy mỗi lần lặp lại bước 2 và 3 thì trị số k tăng lên gấp đôi do đó số lần lặp của hai bước này là  $\log_2 n$  và độ phức tạp của hai bước này tỷ lệ thuận với độ dài n của dãy đã cho. Do đó độ phức tạp của Merge Sort là  $O(n \log_2 n)$ . Trong quá trình sắp xếp trộn, ta không sử dụng thông tin về trạng thái của dãy đã cho, vì vậy trong mọi trường hợp, độ phức tạp của thuật toán này không thay đổi. Đó chính là nhược điểm nổi bật của Merge Sort. Ngoài ra do bản chất của Merge Sort phải phân phối các Run lên các dãy con b, c nên cần thêm không gian nhớ để lưu b, c. Với các dãy có kích thước lớn (hàng vạn) và cực lớn (hàng triệu) trở lên thì đây là nhược điểm lớn.

## 2.14. SẮP XẾP CƠ SỐ (RADIX SORT HAY POST SORT)

### 2.14.1. Mở đầu

Radix Sort có tên gọi khác gần gũi hơn: Post Sort-sắp xếp Bưu điện. Tên gọi Post Sort xuất phát từ một nhận xét thực tế trong công việc phân loại thư của Bưu điện như sau:

Để chuyển khối lượng khổng lồ thư tín từ trung tâm bưu điện đến người nhận ở nhiều nơi khác nhau, bưu điện tổ chức một cơ chế *phân loại thư phân cấp*. Trước tiên họ bỏ các thư gửi tới cùng một tỉnh, hoặc thành phố vào cùng một lô để chuyển đến các tỉnh thành phố tương ứng. Đến lượt, bưu điện các tỉnh, thành lại bỏ các thư gửi tới cùng một quận, hoặc huyện vào chung một lô và chuyển đến quận, huyện tương ứng. Rồi bưu điện các quận, huyện lại xếp các thư gửi đến cùng một phường hoặc xã vào chung một lô để sau đó các thư này được gửi đến các phường, xã tương ứng.

### 2.14.2. Ý tưởng

So sánh với các phương pháp sắp xếp dữ liệu quen thuộc, ta thấy: những phương pháp này dựa vào việc **so sánh giá trị của hai phần tử** trong danh sách, thì Radix Sort không quan tâm đến việc đó mà chỉ quan tâm tới việc **phân loại các phần tử của dãy theo phương pháp Bùn điện**. Vậy cụ thể khi sắp xếp một dãy số nguyên dương thì cái gọi là “phân loại” trong dãy số này là gì? và làm thế nào để sắp dãy số ấy có trật tự (tăng hoặc giảm) bằng thuật toán Post Sort?. Trong bài toán này, “phân loại” dựa vào “danh tính” của các chữ số thuộc các số trong dãy cho trước: chữ số hàng đơn vị, chữ số hàng chục, chữ số hàng trăm,... chữ số hàng vạn, chữ số hàng chục vạn,...

Để dễ hiểu rõ cơ chế làm việc của thuật toán này ta xét việc sắp xếp tăng dãy A dưới đây:

0123 2154 0222 0004 0283 1560 1061 2150

### 2.14.3. Cơ chế làm việc của Post Sort

Quy trình làm việc như sau:

- Các số có chữ số hàng đơn vị là 0 bỏ cùng vào lô  $B_0$ .
- Các số có chữ số hàng đơn vị là 1 bỏ cùng vào lô  $B_1$ .
- .....
- Các số có chữ số hàng đơn vị là 9 bỏ cùng vào lô  $B_9$  (Xem bảng 1)

*Bảng 2.1. Phân lô theo hàng đơn vị*

8	2150									
7	1061									
6	1560									
5	0283									
4	0004									
3	0222									
2	2154	2150				2154				
1	0123	1560	1061	0222	0123	0004				
CS	A	0	1	2	3	4	5	6	7	8
										9

Các lô (ký hiệu  $L_0, L_1, \dots, L_9$ ) dùng để phân loại

Đặt các số có chữ số hàng đơn vị giống nhau vào cùng lô có Index trùng với chữ số hàng đơn vị.

Tiếp tục áp dụng cách thức trên cho các chữ số hàng chục, hàng trăm, hàng ngàn, hàng vạn... thuộc các số đã cho trong dãy. (Xem các bảng tương ứng dưới đây!).

*Kết nối (combined) Bảng 2.2. Phân lô theo hàng chục*

kết quả bảng 1

8	0004										
7	2154										
6	0283										
5	0123										
4	0222										
3	1061										
2	2150		0222			2154	1560				
1	1560	0004	0123			2150	1061		0283		
CS	A	0	1	2	3	4	5	6	7	8	9

Đặt các số có chữ số hàng chục giống nhau vào cùng một lô có Index trùng với chữ số hàng chục.

*Bảng 2.3. Phân lô theo hàng trăm*

*Kết nối (combined)*

kết quả bảng 2.2

8	0283										
7	1061										
6	1560										
5	2154										
4	2150										
3	0123		2154								
2	0222	1061	2150	0283							
1	0004	0004	0123	0222			1560				
CS	A	0	1	2	3	4	5	6	7	8	9

Đặt các chữ số hàng trăm giống nhau vào cùng một lô có chỉ số trùng với chữ số hàng trăm.

Cuối cùng là bảng phân lô theo hàng ngàn (Bảng 2.4)

Bảng 2.4. Phân lô theo hàng ngàn

Kết nối (combined)

kết quả bảng 2.3



8	1560										
7	0283										
6	0222										
5	2154										
4	2150	0283									
3	0123	0222									
2	1061	0123	1560	2154							
1	0004	0004	1061	2150							
CS	A	0	1	2	3	4	5	6	7	8	9

Đặt các số **hàng ngàn** vào các lô có **chỉ số trùng** với các **chữ số hàng ngàn**

Sau đó ta ghép nối các số ở bảng phân lô cuối cùng (bảng 2.4) lại để được dãy có thứ tự (tăng hoặc giảm, xem bảng 2.5). Kết thúc công việc.

Kết nối (combined)    Bảng 2.5. Bảng kết quả sắp xếp dãy A tăng dần

kết quả bảng 4 ta được

bởi Radix Sort

dãy đã cho A xếp tăng dần

(theo index từ thấp đến cao)



8	2154										
7	2150										
6	1560										
5	1061										
4	0283										
3	0222										
2	0123										
1	0004										
CS	A	0	1	2	3	4	5	6	7	8	9

#### 2.14.4. Các bước thực thi thuật toán Post Sort

Bước 1: Khởi trị  $cs=0$ ; //  $cs$ : chữ số để phân loại;  $cs=0 \rightarrow$  chữ số hàng đơn vị;

$cs=1 \rightarrow$  chữ số hàng chục;  $cs=2 \rightarrow$  chữ số hàng trăm;...

Bước 2: Tạo các lô  $B_0, B_1, \dots, B_9$  và khởi trị tất cả chúng là rỗng.

Bước 3: Duyệt từ đầu dãy đã cho đến hết dãy làm:

+ Đặt số  $a_i$  vào lô  $B_k$  với  $k$  là chữ số thứ  $k$  của  $a_i$ .

Bước 4: Ghép các số ở các lô  $B_0, B_1, \dots, B_9$  với nhau (theo trình tự thỏa mãn trật tự xếp của ta) để thu được dãy A có trật tự.

Bước 5: Tăng  $cs=cs+1$ ;

Nếu  $cs < m$  ( $m$ : các chữ số thuộc các số trong dãy đã cho:  $0 \leq m \leq 9$ ) thì quay lại Bước 2 ;

Còn thì dừng;

#### 2.14.5. Cài đặt Post Sort

Từ quy trình làm việc mô tả trên đây của Post Sort, ta thấy luôn luôn phải chèn thêm phần tử mới vào cuối hoặc đầu danh sách (tùy theo sắp tăng hay giảm). Công việc này chỉ thuận tiện khi ta dùng *Danh sách Liên kết đơn (Single Linked List)* để biểu diễn dãy đã cho. Tới chương 4 ta mới nghiên cứu Single Linked List. Vì vậy đến chương 4 chúng ta sẽ trở lại việc cài đặt Post Sort. Dưới đây chỉ cung cấp *hàm phân loại các phần tử* của dãy đã cho (làm cơ sở cho Post Sort).

#### 2.14.6. Hàm phân loại các phần tử

```

int GetDigit(unsign long n, int k) //Hàm tạo các chữ
                                    //số k để phân loại
{
    switch(k)
    {
        case 0: return (n%10);
        case 1: return ((n%10)%10);
        case 2: return ((n/100)%10);
        case 3: return ((n/1000)%10);
        case 4: return ((n/10000)%10);
        case 5: return ((n/100000)%10);
        case 6: return ((n/1000000)%10);
    }
}

```

```

        case 7: return ((n/10000000)%10);
        case 8: return ((n/100000000)%10);
        case 9: return ((n/1000000000)%10);
    }
}

```

## BÀI TẬP CHƯƠNG 2

2.1. Xây dựng sơ đồ (hoặc bảng) dò vết của các thuật toán đã tìm hiểu ở trên khi cân sắp xếp dãy:

44 77 55 99 66 33 22 88 77

2.2. Cài đặt các thuật toán sau:

- a. Interchange sort
- b. Bubble sort
- c. Shake sort

2.3. Thiết kế một chương trình ALL\_IN\_ONE (tất cả trong một) tổ chức theo Menu để người học có thể kiểm thử một thuật toán sắp xếp bất kỳ trong số các thuật toán đã tìm hiểu trong sách này và cả các thuật toán cho tại mục 2.2 ở bài tập. Yêu cầu: mục chọn “Nhập dữ liệu” có 2 thực đơn con để người dùng tùy chọn:

- + Nhập bằng tay
- + Nhập tự động ngẫu nhiên (dãy số tự động phát sinh có các giá trị ngẫu nhiên)

2.4. a. Tại sao Bubble Sort lại chậm?

b. Với giá trị như thế nào của n thì ĐPT của Bubble Sort sẽ là  $O(n^2)$ ?

2.5. a. Bubble Sort, Selection Sort và Insertion Sort đều là các thuật toán có  $O(n^2)$ . Tuy nhiên vẫn có thuật toán nhanh nhất và chậm nhất trong 3 thuật toán nêu trên. Hãy cho biết thuật toán nào chậm nhất và thuật toán nào nhanh nhất?

b. Merge Sort, Quick Sort và Heap Sort đều là các thuật toán có  $O(n \log n)$ . Hãy cho biết thuật toán nào nhanh nhất, thuật toán nào chậm nhất?

2.6. Nếu một thuật toán có  $O(n^2)$  (Bubble Sort, Selection Sort và Insertion Sort) mất 3,1 ms để chạy trên một dãy 200 phần tử thì dự kiến sẽ mất bao lâu để chạy trên một dãy có 400, 40000 phần tử?

2.7. Nếu một thuật toán có  $O(n \log n)$  (Merge Sort, Quick Sort và Heap Sort) mất 3,1 ms để chạy trên một dãy 200 phần tử thì dự kiến sẽ mất bao lâu để chạy trên một dãy có 400, 40.000 phần tử?

2.8. a. Bubble Sort sẽ chạy như thế nào (nhanh hay chậm) trên một dãy:

- Đã sắp xếp?
- Đã sắp xếp theo thứ tự ngược lại?
- b. Câu hỏi tương tự với Merge Sort.
- c. Câu hỏi tương tự với Quick Sort.
- d. Câu hỏi tương tự với Heap Sort.

2.9. Một người đã thiết kế hàm mô tả thuật toán Bubble Sort nhờ dùng template như sau:

```
template<class T> void sort(T* a, int n)
{
    for(int i=1; i<=n; i++)
        for(int j=1; j<=n-1; j++)
            if(a[j-1]>a[j])
                swap(a[j-1], a[j])
}
```

Như ta thấy, thuật toán trên chưa “đủ thông minh”, hãy cải tiến Bubble Sort “đủ thông minh” để nó “tự” ngừng lại khi dãy đã sắp xếp.

2.10. Merge Sort có tính song song, tức là các đoạn con của nó có thể được xử lý đồng thời độc lập với nhau, với điều kiện PC có các bộ xử lý chạy song song. Còn các thuật toán sắp xếp nào mô tả trong chương này có tính song song?

2.11. Một thuật toán sắp xếp được gọi là “ổn định” nếu nó bò qua thứ tự của các phần tử tương đương. Trong các thuật toán đã xét, thuật toán nào là thuật toán “ổn định”?

2.12. Cho dãy số: 44 77 55 99 66 33 22 88 77

Hãy lập bảng dò vết khi thực thi từng thuật toán đã tìm hiểu trong chương này trên dãy đã cho.

2.13. Chứng minh định lý Về sự giới hạn ĐPT đối với các thuật toán sắp xếp nhờ đốichỗ sau: “Không có thuật toán sắp xếp nào dùng cách so sánh các phần tử của dãy có thể có ĐPT trong tình huống xấu nhất lại cao hơn  $O(n \log n)$

- ở đây log là log cơ số 2-

## GỢI Ý HOẶC ĐÁP ÁN

### 2.3. ALL\_IN\_ONE

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
#include <ctype.h>
#include <assert.h>
#define true 1
#define false 0
//=====
template <class T>      // Hàm trao đổi
void swap(T& x, T& y)
{
    T tg=x;x=y;y=tg;
}
//=====
void nhap(int a[],int &n)
{ int i,x;
printf("\nNhap du lieu vao danh sach:");
printf("\n 1. Nhap bang tay");
printf("\n 2. Nhap tu dong va ngau nhien");
printf("\n\n Hay chon 1 hoac 2:");
fflush(stdin);
char ch=getch();
printf("\nNhap so phan tu cua danh sach: ");
fflush(stdin);
scanf("%d",&n);
if(ch=='1')
```

```
{ printf("\nHay nhap %d so: ",n);
  for(i=0;i<n;i++)
  {
    printf("a[%d]:",i); scanf("%d",&a[i]);
  }
}
else
{ randomize();
  for(i=0;i<n;i++)
  { x=random(10*n); a[i]=x; }
}
}

//=====
void view(int a[],int n) // Hàm hiển thị dãy vừa nhập
{ int i;
  printf("Day so vua nhap:\n");
  for(i=0;i<n;i++)
printf("%4d",a[i]);
}
void xem(int a[],int n)
{int i;
  for(i=0;i<n;i++)
printf("%4d",a[i]);
}

//=====
int sorted(int a[],int n) // Hàm kiểm tra xem dãy đã sắp
                           // xếp chưa
{int i;
  for(i=0;i<n-1;i++)
```

```

    if(a[i]>a[i+1]) return false;
    return true;
}
=====

void selectsort(int a[],int n) // Hàm thực thi xếp chọn
{int t,i,j,k,min;
for(i=0;i<n-1;i++)
{ min=a[i];k=i;
for(j=i+1;j<n;j++)
    if(a[j]<min) {k=j;min=a[j];}
    if(k!=i) swap(a[i],a[k]);
}
}

=====

void bubblesort(int a[],int n) // Hàm thực thi xếp nổi bọt
{int t,i,j;
for (i=0; i<n-1; i++)
    for(j=n-1;j>i; j--)
        if(a[j]<a[j-1]) swap(a[j],a[j-1]);
}

=====

void insertsort(int a[],int N, int h=1) // Hàm thực thi
                                         // xếp chèn
{int i,j,int x;
for(i=h;i<N;i++)
{x=a[i];
j=i;
while(j>0 && x<a[j-h])
{ a[j]=a[j-h];           //Lưu phần tử lớn hơn x vào mảng a[j]
}
}

```

```
j=j-h;                                //cách phần tử hiện thời h vị trí
};

a[j]=x;

}

};

//=====

void shellsort(int a[],int N,int step[],int
               buocnhay)          // Hàm thực thi xếp shellsort

{ int s,i,j,h;int x;
for(s=0;s<buocnhay;s++)
{h=step[s];
insertsort(a,N,h);
}
};

//=====

// Hàm phân hoạch

void partition(int a[],int low,int up,int& pivot)
{int pivotval=a[low];
int i=low;
int j=up;
while(i<j)
{while(a[i]<=pivotval && i<up) i++;
while(a[j]>pivotval) j--;
if(i<j) swap(a[i],a[j]);
};
swap(a[low],a[j]);pivot=j;
}

//=====

void quicksort(int a[],int low,int up) // Hàm thực thi
                                         // xếp nhanh
```

```

    {int pivot;
     if(low>=up) return;
     partition(a,low,up,pivot);
     quicksort(a,low,pivot-1);
     quicksort(a,pivot+1,up);
    }
//=====
void heapsort(int a[],int n) // Hàm thực thi HeapSort
{ int i,s,f;int x;
for(i=1;i<n;i++)
{ x=a[i];
  s=i;           //s là nút con, hiện tại trên heap chưa có a[i]
  f=(s-1)/2 la nut cha
  while(s>0 && x>a[(s-1)/2])
  {a[s]=a[(s-1)/2]; //Đặt nút có giá trị < x xuống thấp 1 mức
   (trên cây heap)
   s=(s-1)/2;
  };
  a[s]=x;
};
for(i=n-1;i>0;i--)
{ x=a[i];a[i]=a[0];
  f=0;           //f là nút cha, s là nút con
  s=2*f+1;       //Gán s là nút con bên trái
  if(s+1<i && a[s]<a[s+1]) s=s+1; //Nếu có nút phải lớn
                                         //hơn thì chọn nút phải đó
  while(s<i && x<a[s])
  {a[f]=a[s];      //Thay thế nút cha bằng nút con lớn hơn
   f=s;             //Chuyển đến con lớn tiếp theo
   s=2*f+1;
}
}

```

```
        if(s+1<i && a[s]<a[s+1]) s=s+1;
    };
    a[f]=x;                                //Khóa x chèn vào đúng vị trí f
};
//=====
void mergesort(int a[],int n)
- {int i,j,k,low1,up1,low2,up2; //low: cận dưới và up: cận
   trên của 2 danh sách con
  int size;
  int *Ds_tg=new int[n];
  size=1;                                //Khởi trị: gán size bằng 1
  while(size<n)
  {low1=0;k=0;
   while(low1+size<n)
   { low2=low1+size;
     up1=low2-1;
     up2=(low2+size-1<n)?low2+size-1:n-1;
     for(i=low1,j=low2;i<=up1 && j<=up2;k++)
       if(a[i]<=a[j]) Ds_tg[k]=a[i++];
       else Ds_tg[k]=a[j++];
     for(i <= up1; k++) Ds_tg[k] = a[i++];
     for(j <= up2; k++) Ds_tg[k] = a[j++];
     low1 = up2+1;
   }
   for(i = low1; k<n; i++) Ds_tg[k++] = a[i];
   for(i = 0; i<n; i++) a[i] = Ds_tg[i];
   size *= 2;
}
}
```

```
//=====
void Xepdoicho(int a[],int n) // Hàm xếp đôi chéo
{ int i,j;
for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
        if(a[j]<a[i])
            swap(a[i],a[j]);
}
//=====
void ShakerSort(int a[],int n) // Hàm thực thi ShakerSort
{ int i,j;
int left, right, k;
left=0;
right = n-1;
k = n-1;
while (left < right)
{ for (j = right; j > left; j --)
    if (a[j]< a[j-1])
    {
        swap(a[j], a[j-1]);
        k =j;
    }
    left = k;
for (j = left; j < right; j++)
    if (a[j]> a[j+1])
    { swap(a[j], a[j+1]);
        k = j;
    }
right = k;
}
```

```
}

//=====
void main()
{
    int step[3] = {5,3,1};
    int a[50]; int t,n,i;int x;
    while(true)
    {
        clrscr();
        printf("\n a. Nhập danh sách");
        printf("\n b. Xem dãy số Vua nhập:");
        printf("\n c. Sắp xếp tăng dần bằng Select Sort");
        printf("\n d. Sắp xếp tăng dần bằng Bubble Sort");
        printf("\n e. Sắp xếp tăng dần bằng Insert Sort");
        printf("\n f. Sắp xếp tăng dần bằng Shell Sort");
        printf("\n g. Sắp xếp tăng dần bằng Quick Sort");
        printf("\n h. Sắp xếp tăng dần bằng Heap Sort");
        printf("\n i. Sắp xếp tăng dần bằng Merge Sort");
        printf("\n j. Sắp xếp tăng dần bằng Interchange
              Sort");
        printf("\n k. Sắp xếp tăng dần bằng Shake Sort");
        printf("\n 0. Kết thúc");
        printf("\n Hãy go mot trong cac phim thuoc
              {a...0} de chon viec: ");
        char chon=getch();
        if (chon=='0') break;
        printf("\n");
        switch (chon)
        {
            case 'a': nhap(a,n);break;
            case 'b': view(a,n);break;
            case 'c': selectsort(a,n);
                printf("Day da xep tang dan bang pp Select
                      Sort: \n\n");
                xem(a,n); break;
        }
    }
}
```

```
case 'd': bubblesort(a,n);
            printf("Day da xep tang dan bang pp Bubble
                    Sort:\n\n");
            xem(a,n); break;
case 'e': insertsort(a,n);
            printf("Day da xep tang dan bang pp Insert
                    Sort:\n\n");
            xem(a,n); break;
case 'f': shellsort(a,n,step,3);
            printf("Day da xep tang dan bang pp Shell
                    Sort:\n\n");
            xem(a,n); break;
case 'g': quicksort(a,0,n-1);
            printf("Day da xep tang dan bang pp Quick
                    Sort:\n\n");
            xem(a,n); break;
case 'h': heapsort(a,n);
            printf("Day da xep tang dan bang pp Heap
                    Sort:\n\n");
            xem(a,n); break;
case 'i': mergesort(a,n);
            printf("Day da xep tang dan bang pp Merge
                    Sort:\n\n");
            xem(a,n); break;
case 'j': Xepdoicho(a,n);
            printf("Day da xep tang dan bang pp
                    Interchange Sort: \n\n");
            xem(a,n); break;
case 'k': ShakerSort(a,n);
            printf("Day da xep tang dan bang pp Shake
                    Sort:\n\n");
            xem(a,n); break;
```

```

    }

    printf("\n\n Nhấn phím bất kỳ để tiếp tục");
    getch();
}

}

```

2.4. a. Bubble sort chậm vì nó chỉ hoạt động theo cách thức cục bộ, từng phần tử thuộc dây, mỗi lần chỉ di chuyển 1 vị trí.

b. Với những giá trị n rất lớn hoặc cực lớn (chẳng hạn  $n=10^8$ ) thì  $n(n-1)/2 \approx n^2$ .

Khi đó ĐPT(Bubble Sort)= $O(n^2)$ .

2.5. b. Merge Sort chậm hơn Heap Sort và trong hầu hết các trường hợp Quick Sort nhanh hơn cả.

2.7. Thời gian chạy t tỷ lệ với  $n \log n$ , do đó sẽ tồn tại hằng số c để  $t=c.n \log n$ . Nếu mất  $t=3,1$  ms để sắp xếp  $n=200$  phần tử thì  $3,1=c.(200).\log(200)$ , vì vậy  $c=(3,1 \text{ ms})/(200.\log(200))=0,0155/\log(200)$ . Suy ra với  $n=40.000$ ,  $t=c.n \log n = (0,0155/\log(200))(40.000.\log(40.000))=620.(\log(40.000)/(\log(200)))$ .

Vì  $40.000 = (200)^2$  nên  $\log(40000)=\log(200^2)=2.\log(200)$ .

Do vậy:  $\log(40000)/\log(200)=2$  nên  $t=620*2=1240 \text{ ms}=1,24 \text{ s}$ .

2.8. a. Bubble Sort là thuật toán không nhạy với đầu vào. Nghĩa là nó sẽ thực thi cùng số lần so sánh là  $n.(n-1)/2$  bất kể thứ tự ban đầu của mảng đã cho như thế nào. Vì vậy mảng đã được sắp xếp hay mảng sắp xếp theo trật tự ngược lại thì Bubble sort vẫn chạy rất chậm (đặc biệt với n rất lớn hoặc lớn).

d. Heap Sort hơi nhạy với đầu vào vì vậy nó có thể cần số lần so sánh ít hơn  $\log n$ .

### 2.9. Hàm mẫu <class T>

```

void sort(T*a, int n)
{
    bool sorted=false;
    for(int i==1; i<n && !sorted; i++)
        for(int j=1; j<=n-i; j++)
            { sorted=true;

```

```

        if( a[j-1] > a[j])
        { swap(a[j-1], a[j]);
          sorted=false;
        }
    }
}

```

2.10. Bên cạnh Merge Sort chỉ có Quick Sort là có tính song song. Tiến trình phân hoạch của Quick sort có thể thực thi đồng thời trên các đoạn đã phân hoạch của mảng.

2.11. Quick Sort và Shell Sort là các thuật toán không ổn định.

2.13. (Lời giải bài này phải dùng khái niệm cây “Quyết định”, mời đọc giả xem mục 5.1.3 Chương 5, trước khi đọc mục này!)

Ta dùng *cây quyết định* để chứng minh định lý này (vì cách này giản dị và dễ hiểu). Do thuật toán sắp xếp mảng dùng cách so sánh các phần tử thuộc nó nên mọi nút trên cây quyết định biểu thị điều kiện ( $a[i] < a[j]$ ). Mỗi điều kiện như vậy có 2 kết cục (true hoặc false). Vì vậy cây quyết định là một cây nhị phân (về định nghĩa, nó cũng đã là cây nhị phân rồi!). Và do cây phải có tất cả các kiểu sắp xếp có thể có nên nó phải có ít nhất  $n!$  lá (dãy cần sắp xếp có  $n$  phần tử). Do vậy độ cao tối thiểu của cây quyết định là  $\log(n!)$ . Trong trường hợp xấu nhất, số lần so sánh do thuật toán sắp xếp thực hiện phải bằng độ cao của cây quyết định. Do vậy ĐPT trong trường hợp xấu nhất của các thuật toán sắp xếp trong trường hợp xấu nhất là  $O(\log n!)$ .

Theo định lý Stirling ta có:

$$n! = \sqrt{2\pi n} \left( \frac{n}{e} \right)^n$$

vì vậy:

$$\log n! \approx \log \left[ \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \right] \approx \log(n^n) \approx n \log n \text{ Chứng minh xong!}$$

(ở đây log là  $\log_2$ )

## Chương 3

# MỘT SỐ CHIẾN LƯỢC THIẾT KẾ THUẬT TOÁN

*"Thiết kế không chỉ là chúng sẽ trông như thế nào mà thiết kế phải là chúng làm việc ra sao."*

*Steve Jobs*

Có một số cách phổ dụng thiết kế thuật toán mà các nhà thuật toán học còn gọi là các chiến lược thiết kế thuật toán. Chúng ta cũng đã biết không có chiến lược nào là vạn năng, tức không có chiến lược nào có thể áp dụng để xây dựng thuật toán cho tất cả các loại bài toán thực tế, cụ thể. Các cách phổ dụng này sẽ trình bày sau đây đã được áp dụng trong thực tế để thiết kế thuật toán cho một lớp khá nhiều các bài toán.

Dưới đây ta sẽ lần lượt trình bày các chiến lược đó.

### 3.1. CHIẾN LƯỢC CHIA ĐỂ (DỄ) TRỊ

Cách chia để trị (Divide and Conquer) còn có các tên gọi khác là *Stepwise Refinement* (tinh chế từng bước) hoặc *Top-Down Design* (thiết kế trên xuống).

#### 3.1.1. Ý tưởng chủ đạo của chiến lược này

Chia bài toán cần giải (mà chưa biết hoặc việc tìm thuật toán cho nó phức tạp) thành các bài toán con. Đến lượt các bài toán con ấy lại được phân chia thành các bài toán con nhỏ hơn, rồi tiếp tục cách thức đó cho tới khi thu được các bài toán con đã có thuật toán hoặc việc tìm và thiết kế thuật toán cho chúng dễ dàng hơn. Các bài toán con trong quá trình phân hoạch thuộc cùng loại (lớp) với bài toán đã cho. Ở bước kết thúc quy trình trên, ta kết hợp các nghiệm của những bài toán con để thu được nghiệm của bài toán lớn hơn và cuối cùng là nghiệm của bài toán cần giải. Để thực thi chiến lược này người ta hay dùng phương pháp đệ quy (sẽ trình bày ở mục 3.2 dưới đây).

#### 3.1.2. Một số minh họa cho chiến lược “chia để (dễ) trị”

##### a. Minh họa 1

Một minh họa kinh điển cho chiến thuật “chia để (dễ) trị” là bài toán Binary Search đã trình bày kỹ ở mục 2.2. Trong thuật toán này thay vì dò tìm trên toàn bộ dãy, người ta chia đôi dãy nhờ một phần tử làm mốc ở giữa dãy.

(Nghĩa là dãy ban đầu gồm 3 dãy con: dãy thứ nhất gồm các phần tử ở bên trái mốc, dãy con thứ hai gồm các phần tử ở bên phải mốc, dãy con thứ ba chỉ có độc nhất một phần tử: đó là phần tử giữa). Vấn đề đặt ra là xác định xem một khoá k cho trước có thuộc dãy đã cho không? Với cách làm trên, ta chỉ cần xác định xem *k* thuộc dãy con nào. Nếu ai đã học “Lý thuyết thông tin Shannon” thì đều biết rằng với “chiến thuật khôn lỏi” như trình bày thì mỗi lần chia đôi dãy ta đã làm giảm độ bất định của việc dò tìm khoá k đi 2 lần và làm tăng lượng thông tin tìm kiếm (sự hiểu biết) lên 2 lần (đó là nói một cách áng chừng, còn chính xác có thể là một lượng nhỏ hơn 2 một chút!). Tiếp tục ta lại chia đôi các dãy con hiện hành bằng “chiến thuật” trên (tức là tiếp tục *thu hẹp “không gian” dò tìm đi nữa đến khi không thể thu hẹp được nữa - khi dãy chỉ còn duy nhất một phần tử - thì dừng*)

### b. Minh họa 2

Chiến lược “chia để (dễ) trị” còn được thể hiện rất rõ trong thuật toán Quick Sort đã xét ở mục 2.11. Ở thuật toán đó, thay vì sắp xếp trên toàn dãy đã cho, người ta tiến hành sắp xếp trên các dãy con sinh ra từ dãy ban đầu nhờ cách phân hoạch nó như sau. Chia dãy đã cho thành 3 dãy con dựa vào phần tử mốc M chọn tùy ý thuộc dãy đã cho:

- Dãy con thứ nhất: Gồm các phần tử có giá trị không lớn hơn M:  $a_k \leq M$  ( $k=1,2,\dots,j$ );

- Dãy con thứ hai: Gồm chỉ phần tử mốc M (trong Quick sort không nhất thiết mốc phải là phần tử ở giữa dãy đã cho):  $a_j = M$  ( $j=1,2,3,\dots,n$ ; n: độ dài dãy ban đầu);

- Dãy con thứ ba:  $a_k > M$  ( $k=j, j+1, \dots, n$ );

Vì ta coi *dãy có một phần tử độc nhất cũng là dãy đã có trật tự*, nên ta chỉ quan tâm tới việc sắp xếp dãy con thứ 1 và 3. Nếu dãy 1 và 3 cũng chỉ có 1 phần tử thì dãy đã cho hiển nhiên cũng có trật tự. Còn nếu các dãy con 1, 3 gồm nhiều hơn một phần tử thì để dãy đã cho có trật tự ta cần sắp xếp dãy con 1 và 3. Đến đây ta lại áp dụng quy trình phân hoạch các dãy con 1, 3 như đã làm với dãy đã cho. Tiếp theo lại sắp xếp các dãy con hiện hành theo cách trên... Vì thao tác “xếp” dãy con luôn lặp lại (khi mà chưa xét hết dãy ban đầu) nên trong bài toán này, đệ quy lại được áp dụng.

Ở minh họa này, một lần nữa ta lại thấy “*phương trâm*” làm việc là: *thu hẹp không gian trạng thái cần thao tác để số thao tác ít hơn, dễ thực hiện hơn*.

### 3.2. CHIẾN LƯỢC ĐỆ QUY

Chiến lược đệ quy (Recursion) thường được dùng trong các ngành khoa học khác (nói riêng trong toán học) và khoa học máy tính.

#### 3.2.1. Đệ quy là gì?

Một khái niệm  $K_n$  được định nghĩa theo đệ quy nếu trong định nghĩa  $K_n$  lại sử dụng ngay chính khái niệm  $K_n$ .

Nói một cách khác (tổng quát hơn): Một đối tượng là **đệ quy** nếu nó **được định nghĩa qua chính nó hoặc qua một đối tượng cùng loại** với nó bằng **phép quy nạp**.

*Ví dụ 3.1.* Mô hình thực tế của đệ quy: hãy treo hai chiếc gương đối diện nhau. Khi ấy hình ảnh của chiếc gương này trong chiếc gương kia là đệ quy. Và nếu thời gian là vô tận thì ta cũng thấy một dãy vô hạn hình ảnh hai chiếc gương trong nhau.

*Ví dụ 3.2.* Định nghĩa số tự nhiên:

- + Số 0 là số tự nhiên
- +  $N$  là số tự nhiên nếu  $N-1$  là số tự nhiên.

*Ví dụ 3.3.* Ta gán cho ký hiệu  $!$  tên gọi là “phép tính giai thừa” (nói gọn là “giai thừa”) và  $k!$  hiểu là giai thừa của số nguyên dương  $k$ . Ta có định nghĩa đệ quy của giai thừa như sau:

$$0!=1;$$

$$1!=1;$$

$$2!=2 \times 1!;$$

$$3!=3 \times 2!;$$

.....;

$$k!=k \times (k-1)!$$

#### 3.2.2. Đệ quy trong khoa học máy tính

Như trên đã nói trong lập trình giải các bài toán lớn trên máy tính, để thực thi chiến lược “Chia để trị” người ta hay dùng đệ quy.

Trong lập trình, *một modul chương trình (modul=đơn nguyên, có thể là hàm hoặc thủ tục)* được gọi là **đệ quy** nếu nó gọi tới chính nó.

*Ví dụ 3.4.* Ta thiết kế một hàm để tính giai thừa  $n!$  ( $n$ : nguyên dương)

```
long int gaiithua(int n)
{
    if (n==0 || n==1) return 1;
    else return n*gaiithua(n-1);
}
```

đệ quy (Hàm gaiithua *gọi tới chính nó!*)

Dưới đây là cơ chế hoạt động của hàm đệ quy gaiithua: để xác định, chẳng hạn ta cần tính  $4!$  nhờ hàm gaiithua() xây dựng ở trên. Khi có lời gọi hàm gaiithua(4) thì hàm này sẽ hoạt động như sau:

+ Trước tiên máy ghi nhớ:  $\text{gaiithua}(4)=4*\text{gaiithua}(3);$

*Và thực hiện tính gaiithua(3);*

+ Kế tiếp máy ghi nhớ  $\text{gaiithua}(3)=3*\text{gaiithua}(2);$

*Rồi máy thực hiện tính gaiithua(2);*

+ Tiếp theo máy ghi nhớ  $\text{gaiithua}(2)=2*\text{gaiithua}(1);$

*Rồi máy đi tính gaiithua(1)=1\*gaiithua(0);*

+ Theo định nghĩa  $\text{gaiithua}(0)=1$  và người ta gọi đây là điểm **dừng** của đệ quy. *Và máy sẽ quay ngược lại tính:*

$\text{gaiithua}(1)=1*\text{gaiithua}(0)=1;$

$\text{gaiithua}(2)=2*\text{gaiithua}(1)=2;$

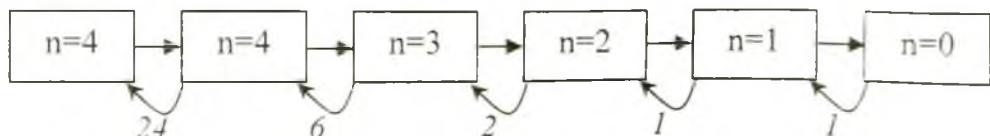
$\text{gaiithua}(3)=3*\text{gaiithua}(2)=6;$

$\text{gaiithua}(4)=4*\text{gaiithua}(3)=24;$

Vậy kết quả  $4!=24.$

*Sơ đồ dò vết hàm gaiithua đệ quy:*

main()    gaiithua(4)    gaiithua(3)    gaiithua(2)    gaiithua(1)    gaiithua(0)



### 3.2.3. So sánh đệ quy và lặp

Lặp	Đệ quy
1. Giống nhau: Lặp lại thực thi một hoặc một vài thao tác nào đó.	1. Giống nhau: Lặp lại thực thi một hoặc một vài thao tác nào đó.
2. Khác nhau:	2. Khác nhau:
Sử dụng lặp tường minh, kết thúc lặp khi điều kiện lặp sai.	Sử dụng lặp bằng lời gọi hàm liên tiếp. Dừng lặp khi tới điểm dừng (điểm neo).
Vòng lặp thay đổi giá trị biến điều khiển lặp cho đến khi giá trị ấy làm cho điều kiện lặp sai.	Lời gọi hàm liên tiếp để làm cho việc tính hàm sau đơn giản hơn lời gọi hàm trước cho đến khi gặp trường hợp đơn giản nhất là giá trị của hàm ứng với điểm dừng.
Với một số bài toán thực tế cụ thể, dùng cấu trúc lặp số lệnh trong code khá nhiều. (Ví dụ bài toán tháp HaNoi)	Với một số bài toán thực tế cụ thể, dùng đệ quy, bản code sẽ ngắn gọn hơn lặp khá nhiều (số lệnh giảm xuống rất đáng kể)
Chiếm dụng bộ nhớ ít hơn đệ quy (với cùng một bài toán).	"Ngốn" bộ nhớ nhiều hơn lặp: vì mỗi lần gọi hàm, hệ điều hành lại cấp phát một vùng nhớ cho lần gọi hàm đó và tất nhiên tốn cả thời gian hơn lặp.

Từ bảng so sánh trên ta kết luận: Không phải bất cứ bài toán nào cũng dùng đệ quy (tức là *không nên quá lạm dụng đệ quy*). Sau này ta sẽ thấy, một chiến lược thường được dùng hơn đó là chiến lược Quy hoạch động (còn gọi là “Bottom-Up Strategies”)

**Ví dụ 3.5. (Bài toán Chuyển n đĩa Kinh điển):** Hãy chuyển đĩa thuộc chồng đĩa có n đĩa từ cọc 1 sang cọc 2 với quy luật sau:

Mỗi lần chỉ chuyển một đĩa (đĩa nhỏ luôn nằm trên đĩa lớn ở mọi cọc (hoặc ngược lại-do cách sắp xếp)).

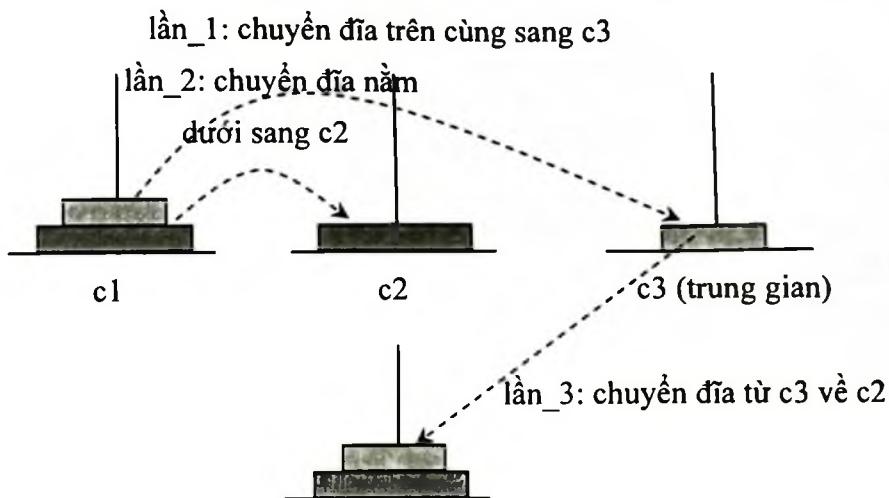
Được phép dùng cọc thứ 3 làm cọc trung gian.

Bài toán trên gọi là bài toán tháp Hà Nội (HaNoi Tower).

a. Phân tích bài toán (xem các hình 3.6 và 3.7 dưới đây):

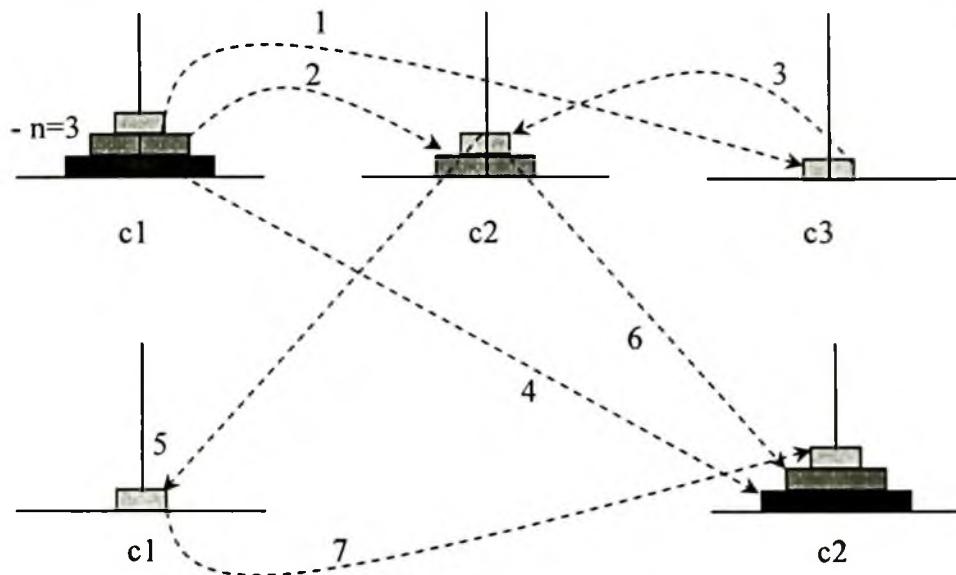
- Trước tiên ta xét trường hợp đơn giản nhất:  $n=1$  (có 1 đĩa), ta chỉ việc chuyển trực tiếp đĩa từ cọc 1 sang cọc 2.

-  $n=2$



Hình 3.1. Sơ đồ chuyển 2 đĩa từ cọc c1 sang cọc c2

Từ sơ đồ ta thấy với số đĩa  $n=2$  thì số lần chuyển đĩa (để hoàn thành yêu cầu bài toán) là  $3=4-1=2^2-1$



Hình 3.2. Sơ đồ chuyển 3 đĩa từ cọc c1 sang c2

với  $n=3$  ta phải làm 7 lần chuyển,  $7=8-1=2^3-1$ .

V.v...

*Tổng quát:* Bằng quy nạp ta có thể chứng tỏ số lần chuyển  $n$  đĩa từ cọc  $c1$  sang cọc  $c2$  là  $2^n - 1$ .

### b. Cài đặt

Như ở bảng so sánh trên đã nói, đây là một trong các bài toán thực tế mà nếu dùng lặp thì code rất dài.

Song dùng đệ quy cho bài này thì code rút ngắn được khá nhiều, cụ thể code gồm 3 lệnh chủ chốt mô tả việc chuyển đĩa sau đây:

- Pha (giai đoạn) thứ nhất chuyển  $n-1$  đĩa từ cọc  $c1$  sang cọc  $c3$
- Pha thứ hai chuyển 1 đĩa từ cọc  $c1$  sang cọc  $c2$
- Pha cuối cùng chuyển  $n-1$  đĩa còn lại từ cọc  $c3$  về cọc  $c2$

Ta gọi đệ quy hàm chuyển đĩa để lặp lại 3 lệnh chủ chốt trên cho tới khi chuyển hết đĩa thì dừng.

Dưới đây là code của bài toán:

```
#include<stdio.h>
#include<conio.h>
int count=0;
void chuyendia(int n, int c1, int c2, int c3)
{
    if(n==1)
        { printf("\n Dia tu coc %d duoc chuyen sang coc
                 %d.\n",c1,c2);
        count++;
    }
    else
    {
        chuyendia(n-1,c1,c3,c2);
        chuyendia(1,c1,c2,c3);
        chuyendia(n-1,c3,c2,c1);
    }
}
```

```

main()
{
    int n;clrscr();
    printf("\n Cho biet so dia can chuyen:");
    scanf("%d",&n);
    chdia(n,1,2,3);
    printf("\n Viec chuyen %d dia da xong!",n);
    printf("\n");
    printf("\n So lan chuyen dia=%d",count);
    getch();
}

```

**Lưu ý:**

Khi test chương trình chúng ta không nên thử với số đĩa lớn (vì với cấu hình máy cho sinh viên thực hành, chương trình chạy rất lâu với số đĩa lớn), trong giờ thực hành chúng ta chỉ nên thử với tối đa số đĩa  $n=8$  là cùng.

Giả sử có một cơ chế máy (PC hoặc ROBOT) một lần chuyển 1 đĩa mất  $1/100$  giây và số đĩa cần chuyển là 64. Ta hãy xem để chuyển số đĩa này phải tốn bao nhiêu thời gian.

Ta gọi  $t$  giây là thời gian chuyển một đĩa, vậy tổng thời gian chuyển  $n=64$  đĩa là:

$$(2^n - 1) * t = (2^{64} - 1) * t = 1,84E+19 * t$$

(trong đó:  $1,84E+19 = 1,84 \cdot 10^{19}$ )

Một năm có:  $365 \cdot 24 \cdot 3600$  giây =  $31536000$  giây.

Vậy số năm để chuyển 64 đĩa từ cọc 1 sang cọc 2 là:  $(1,84 \cdot 10^{19} / 31536000) * t$  (năm)

Theo giả thiết đầu bài  $t=1/100$  giây, thay vào công thức trên ta được:  $5,8 \cdot 10^9$  năm → tức là 5,8 tỷ năm!

Tương truyền kể rằng: Vào thế kỷ 18 có hai du khách châu Âu tới thăm Hà Nội. Hai người này vào một ngôi chùa ở Hà Nội thì thấy một vài nhà sư đang ngồi chuyển các đĩa gỗ từ cọc một sang cọc hai. Hai khách du lịch ấy hỏi các nhà sư: "Chúng tôi muốn chuyển 64 đĩa thì mất bao lâu?". Các nhà sư trả lời ngay: "Phải tới ngày tận thế mới chuyển xong!". Hai ông là hai nhà toán học, sau khi về nước các ông đã mô tả lại việc trên thành bài toán tháp Hà Nội nổi tiếng.

### 3.2.4. Nguyên tắc thiết kế đệ quy

- Đệ thiết kế đệ quy, trước hết cần phải chỉ ra được **điểm dừng** (còn khá nhiều tên gọi khác: điểm neo, trường hợp **cá biệt**, trường hợp **suy biến**, trường hợp **cơ sở** (trong khi đó các sách của Anh, Mỹ chỉ có hai từ: điểm Stop hoặc trường hợp Basic-Basic Case)).

- Điểm dừng là **điểm kết thúc đệ quy**.
- Sau đó chúng ta sử dụng các dữ liệu và thông tin của bài toán tìm ra mối liên hệ giữa trường hợp cá biệt của bài toán với giá trị kế tiếp cần tính. Tiếp tục cách làm đó cho tới khi tìm được cách tính nghiệm của bài toán cần giải theo bước kè trước bước hiện hành.
- Một số bài toán có một basic case (điểm dừng), song một vài bài toán khác lại có thể có nhiều hơn một basic case.

#### Ví dụ 3.6

a. Trong bài toán tính giai thừa, điểm dừng là  $0!=1$ ; mối quan hệ giữa giá trị ở bước hiện hành với giá trị ở bước ngay trước nó được cho bởi:

$$\text{giaithua}(k) = k * \text{giaithua}(k-1);$$

b. Trong định nghĩa đệ quy dãy Fibonacci với độ dài cho trước n thì có 2 hai trường hợp cơ sở:

$$\text{Fibo}(n) = \begin{cases} 0 & \text{nếu } n=0 \leftarrow \text{basic case} \\ 1 & \text{nếu } n=1 \leftarrow \text{basic case} \\ \text{Fibo}(n-1) + \text{Fibo}(n-2) & \text{nếu } n>1 \end{cases}$$

c. Bài toán tính giá trị hàm **Ackermann A(m,n)** cũng có 2 trường hợp cơ sở (basic). Hàm này được định nghĩa bởi hệ thức hồi quy sau:

$$A(m,n) = \begin{cases} n+1 & \text{nếu } m=0; \leftarrow \text{basic case} \\ A(m-1,1) & \text{nếu } n=0 \text{ và } m>0 \leftarrow \text{basic case} \\ A(m-1, A(m, n-1)) & \text{nếu } m>0, n>0 \end{cases}$$

### 3.2.5. Các loại đệ quy

Đệ quy có các loại sau:

- Đệ quy tuyến tính (đã có vài ví dụ ở trên);
- Đệ quy phi tuyến;
- Đệ quy nhị phân;
- Đệ quy tương hỗ;

### a. Đệ quy tuyến tính

Các hàm đệ quy tuyến tính là các hàm được định nghĩa bởi cấu trúc sau:

+ *Cấu trúc cú pháp của đệ quy tuyến tính (mã giả)*

```
void <Function_Name>
{if (điều_kiện_dùng)
{
    <hàm trả về giá trị cần thiết hoặc dừng lại>
}
else
{
    < thực thi một số lệnh>
    < gọi đệ quy function_Name>;
}
```

*Ví dụ 3.7:*

Tính tổ hợp chập k của n phần tử theo công thức:  $\frac{n!}{(n-k)! * k!}$

Bài toán này quy về việc phải viết hàm *đệ quy tuyến tính* tính giai thừa, rồi dùng hàm ấy để tính tổ hợp chập k theo công thức đã cho. Code của bài này như sau:

```
#include<stdio.h>
#include<conio.h>
long gth(int n)
{ if (n==0 || n==1)  return 1;
  else return (n*gth(n-1));
}
main()
{ int n,k; clrscr();
  printf("Nhập k:"); scanf("%d", &k);
  printf("Nhập n(bất buộc n>k):"); scanf("%d", &n);
```

```

    printf("To hop chap %d cua %d phan
           tu=%ld", k, n, gth(n) / (gth(k)*gth(n-k)));
    getch();
}

```

### b. Đệ quy phi tuyến

Đó là các hàm đệ quy mà *lời gọi đệ quy nằm trong thân vòng lặp*. Đệ quy phi tuyến có cấu trúc sau:

+ *Cấu trúc (mã giả)*

```

void <Function_Name>
{
    Lặp
    {
        <thực thi một số lệnh>
        if (điều_kiện_dùng)
            { <thực thi một số lệnh> }
        else { <gọi đệ quy Function_Name> }
    } //Hết lặp
}

```

+ *Ví dụ 3.8:* Cho dãy  $\{X_n\}$  được định nghĩa bởi công thức truy hồi sau:

$X_0=1;$

$X_n=n^2X_0+(n-1)^2X_1+(n-2)^2X_2+\dots+1^2X_{n-1}$  với  $n \geq 1$

+ *Dưới đây là hàm đệ quy phi tuyến tính Xn*

```

unsigned long Xn(int n)
{
    unsigned long sum=0; int i;
    if (n==0)    return 1;
    for (i=0; i<n; i++)
        sum=sum+sqrt(n-i)*Xn(i);
    return sum;
}

```

### c. Đệ quy tương hỗ (Mutual Recursion)

Nếu có hai hoặc nhiều hơn các hàm gọi lẫn nhau thì ta gọi đó là đệ quy tương hỗ.

#### + Cấu trúc (mã giả)

```

void Function_name_1(...);
void Function_name_2(...);

void Function_name_1(...)
{   <thực thi các lệnh>;
    <gọi Function_name_2(...);>
}

void Function_name_2(...)
{   <thực thi các lệnh>;
    <gọi Function_name_1(...);>
}

```

*Ví dụ 3.9:* Một ví dụ khá hay và điển hình minh họa cho đệ quy tương hỗ là hàm đệ quy vẽ đường Hilbert. Đường Hilbert được nhà toán học Đức David Hilbert công bố 1891. Độ dài của nó tại bước lặp thứ n là:

$$L_n = 2^n - \frac{1}{2^n}$$

Phần tử cơ bản để tạo nên đường Hilbert là hình cái tách (cup), người ta cũng nói: đường Hilbert sơ khởi là “chiếc tách nhỏ” đơn lẻ - xem hình 3.3a. Hình 3.3b là một *mạch cơ sở* (*basic vein*) của đường Hilbert. Đó là kết quả của các phép quay và ghép nối các “cái tách nhỏ” với nhau. Mạch cơ sở của đường Hilbert gọi là “cái tách lớn” (Hình 3.3b).



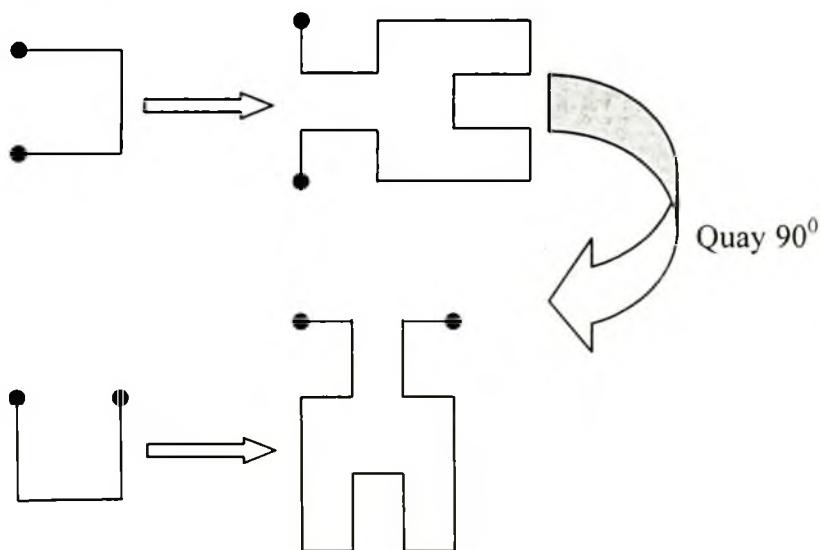
Hình 3.3. Phần tử cơ bản và mạch cơ sở tạo nên đường Hilbert

“Cái tách lớn” của đường Hilbert được tạo nên từ 4 “tách” nhỏ hơn bằng 3 mồi nôi được đánh dấu bởi các chấm đen trên hình 3.3b. Và bất kỳ mỗi lần lặp lại nào của mạch cơ sở cũng là sự *thay đổi vị thế* của các “cái tách nhỏ” theo luật mô tả ở sơ đồ sau (hình 3.4):

$$\begin{aligned}\square &\Rightarrow \square \downarrow \square \rightarrow \square \uparrow \square \\ \square &\Rightarrow \square \rightarrow \square \downarrow \square \leftarrow \square \\ \square &\Rightarrow \square \uparrow \square \leftarrow \square \downarrow \square \\ \square &\Rightarrow \square \leftarrow \square \uparrow \square \rightarrow \square\end{aligned}$$

Hình 3.4. Bốn quy luật thay đổi vị trí của “cái tách nhỏ”

Cả 4 quy luật (rules) trên thực chất chỉ là một: đó là quay “cái tách nhỏ” (và tách lớn)  $90^0$  (hình 3.5):



Hình 3.5. Phép quay “cái tách nhỏ” để tạo mạch cơ sở

Song, thay vì quay mạch cơ sở của đường Hilbert liên tiếp  $90^0$  ta có thể quay nó đi một góc  $90^0$ ,  $180^0$  và  $270^0$ . Ta ký hiệu mạch cơ sở của đường Hilbert là A;

- + Thao tác quay A  $90^0$  là D;
- + Thao tác quay A  $180^0$  là C;
- + Và thao tác quay A  $270^0$  là B;

Các thao tác trên được mô tả bằng các hàm trên ngôn ngữ C/C++. Đến đây ta có cài đặt đệ quy tương hỗ cho bài toán này như sau:

+ *Cài đặt đệ quy tương hỗ vẽ đường Hilbert*

```
#include <conio.h>
#include <graphics.h>

int h;
void A (int);
void B (int);
void C (int);
void D (int);
void A(int i)
{
    if (i>=0)
    {
        D(i-1); linerel(-h, 0);
        A(i-1); linerel(0, -h);
        A(i-1); linerel(h, 0);
        B(i-1);
    }
}
void B(int i)
{
    if (i>=0)
    {
        C(i-1); linerel(0, h);
        B(i-1); linerel(h, 0);
        B(i-1); linerel(0, -h);
        A(i-1);
    }
}
```

$$\left\{ \begin{array}{l} \text{Đệ quy tương hỗ!} \end{array} \right.$$

```
void C(int i)
{
    if (i>=0)
    {
        B(i-1); lineref(h, 0);
        C(i-1); lineref(0, h);
        C(i-1); lineref(-h, 0);
        D(i-1);
    }
}
void D(int i)
{
    if (i>=0)
    {
        A(i-1); lineref(0, -h);
        D(i-1); lineref(-h, 0);
        D(i-1); lineref(0, h);
        C(i-1);
    }
}
void Hilbert()
{
    h = getmaxx() / 99;
    moveto(getmaxx()-130, getmaxy()-50);
    A(5);
}
void main()
{
    int gr_drive = DETECT, gr_mode;
```

```

initgraph(&gr_drive, &gr_mode, "D:\\TC\\BGI");
//Nếu ở PC của bạn, folder TC để tại ổ đĩa khác thì bắt buộc bạn phải đổi lại tên
//đĩa (ở đây tên ổ đĩa là D)!

    setcolor(YELLOW);

    rectangle(0, 0, getmaxx(), getmaxy());

    Hilbert();

    getch();

    closegraph();
}

```

*Ví dụ 3.10:* Trình bày một ví dụ đơn giản hơn do vậy sẽ dễ hiểu hơn về đệ quy tương hỗ, đó là tính các hàm sinx và cosx. Có nhiều cách để tính giá trị của chúng, nhưng có một cách đơn giản nhất (song không phải là hiệu quả nhất) là dùng đệ quy tương hỗ để tính gần đúng sinx và cosx nhờ khai triển Taylor lấy gần đúng đến số hạng thứ hai:

$$\sin x \approx x - x^3/6; \cos x \approx 1 - x^2/2;$$

Sau đây là đệ quy tương hỗ để tính gần đúng sinx và cosx với sai số 5/1000

```

float si(float x)
{
    if(-0.005<x && x<0.005) return x-x*x*x/6; // các hàm
                                                // này đã sử dụng hai công thức
    Return 2*si(x/2)*c(x/2); // lượng giác:
}
                                                // sin2x=2sinxcosx, cos2x=1-2(sinx)^2
float cosi(float x)
{
    if((-0.005<x && x<0.005)      return 1.0-x*x/2;
    return 1-2*si(x/2)*si(x/2)
}

```

### 3.3. ĐỆ QUY VÀ QUAY LẠI (RECURSION AND BACKTRACKING)

#### 3.3.1. Đặc trưng chủ yếu của chiến lược quay lui

Trong thực tế để giải hiệu quả (có thể hiểu theo nghĩa: thời gian tính toán ít nhất, quy trình tìm lời giải đơn giản - không phức tạp) một số bài toán không tuân theo luật tính toán cố định mà dùng chiến lược “*Thử - Sai và Quay lui*”

(*Try - Error and Backtracking; phương pháp này cũng còn có tên gọi là Vết cạn*). Tinh thần chủ đạo và là nét đặc trưng chủ yếu của chiến lược quay lui là **Thử và Sai**.

- + Tổ chức không gian tìm kiếm nghiệm (lời giải) bài toán thành không gian **phân cấp** (*cây tìm kiếm* - ta sẽ nghiên cứu kỹ ở Chương 5).
- + Dùng hàm đệ quy để dò tìm từ cấp thấp đến cấp cao thuộc cây tìm kiếm đó.
- + Tại mỗi cấp xét (thử và sai) tất cả các khả năng có thể lựa chọn nghiệm (thỏa mãn yêu cầu đặt ra): Nếu có một lựa chọn chấp nhận được thì ghi nhớ các thông tin của nó và tiến hành thử ở cấp cao hơn thuộc cây tìm kiếm.
- + Ngược lại không có lựa chọn nào chấp nhận được thì *Quay lui* lại bước trước rồi tiến hành Thử và Sai với các trạng thái còn lại và xóa bỏ các thông tin vừa chọn ở bước vừa rồi.
- + Người đầu tiên đề ra thuật ngữ quay lui vào khoảng những năm 1950 là nhà toán học Mỹ D.H.Lehmer.

### 3.3.2. Minh họa kinh điển cho thuật toán quay lui: *Bài toán Tám quân hậu*

a. **Lịch sử của thuật toán:** Bài toán tám quân hậu lần đầu tiên được nêu ra vào năm 1848 bởi kỳ thủ Max Bezzel. Sau đó nhiều nhà toán học, trong đó là Gauuss và George Cantor công bố một số công trình về bài toán này. Năm 1850 Franz Nauck đưa ra các lời giải đầu tiên về bài toán ấy và tổng quát hóa nó thành bài toán n quân hậu. Năm 1874 S.Gunther công bố lời giải bài toán trên bằng cách dùng định thức và J.W.L Glaisher hoàn thiện phương pháp định thức giải bài toán tám hậu của S.Gunther.

b. **Phát biểu bài toán:** Tìm cách đặt 8 quân hậu trên bàn cờ vua 8x8 sao cho chúng không ăn (chiếu) được nhau.

Muốn hiểu được cách giải bài toán này, trước hết chúng ta phải hiểu rõ một số quy luật cơ bản của bàn cờ vua 8x8.

c. **Một số đặc trưng nổi bật trên bàn cờ vua 8x8.** (xem hình 3.6)

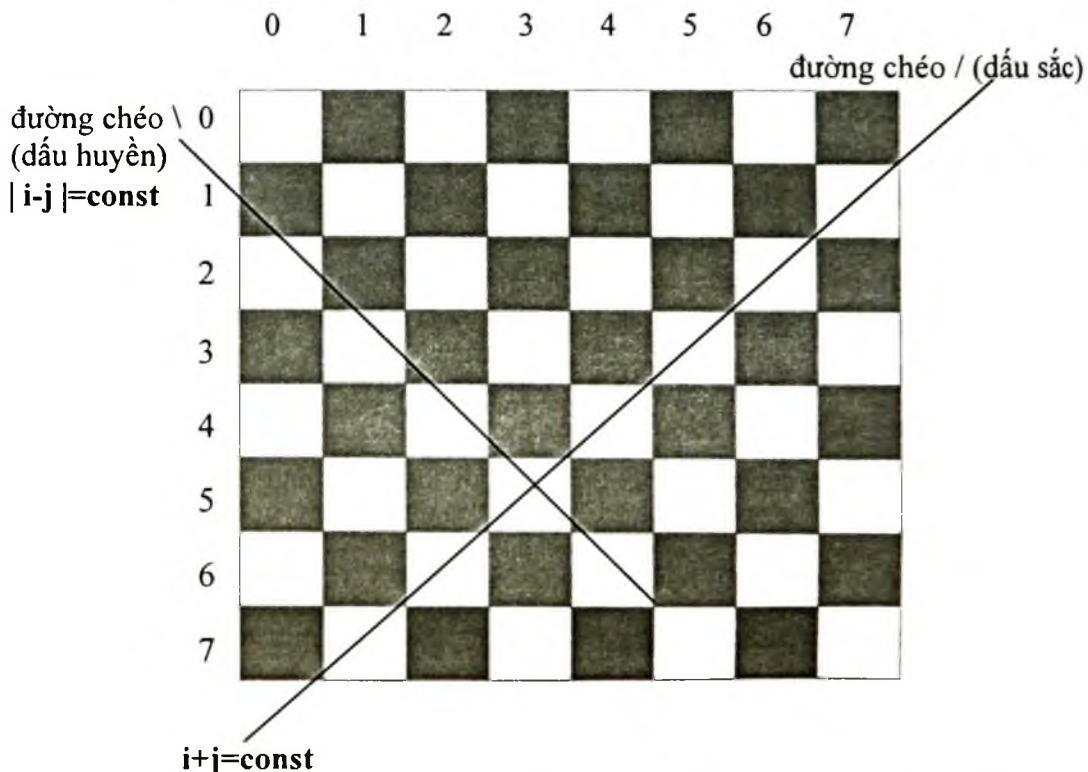
- Mọi ô nằm trên đường chéo “úp” (dấu sắc /) đều có  $tổng i+j = const$  ( $=10$ ; i chỉ số hàng, j chỉ số cột trên bàn cờ vua). Có cả thảy 15 đường chéo ấy được đánh số từ 0 → 14.
- Mọi ô trên đường chéo “ngửa” (dấu huyền \) đều có hiệu  $| i-j | = const$  ( $=2$ ). Có tất cả 15 đường chéo như vậy được đánh số từ -7 đến 7.

- Có nhiều cách để giải bài toán này, song các cách ấy đều dài hơn cách sau đây. Nếu lợi dụng hai đặc trưng nổi bật nêu trên thì cách giải dưới đây là hay hơn cả và ngắn hơn cả. Trong cách này chúng ta dùng kỹ thuật đánh dấu. Để làm việc này chúng ta chỉ cần dùng 3 mảng (không kể mảng bắt buộc phải có là mảng  $b[i][j]$  biểu diễn bàn cờ): một mảng đánh dấu hàng, một mảng đánh dấu đường chéo dấu huyền và một mảng đánh dấu đường chéo dấu sắc.

#### d. Thực chất cơ chế hoạt động của backtracking khi xếp hậu

- Theo luật cờ vua ta đã biết: một quân hậu có thể ăn quân hậu bất kỳ khác trên cùng hàng, cùng cột, cùng đường chéo dấu sắc (ký hiệu tắt là  $dc\_sac$ ) và cùng đường chéo dấu huyền (ký hiệu tắt là  $dc\_huyen$ ). Từ đó ta thấy trên mỗi hàng chỉ có thể đặt một quân hậu, và các điều kiện an toàn hay không an toàn (hậu có thể ăn nhau) được biểu thị bởi các điều kiện sau:

- $h[i]=0$ : không có quân hậu nào trên hàng  $i$ ;  $h[i]=1$ : có quân hậu trên hàng  $i$
- $dc\_sac[i+j]=0$ : không có quân hậu trên đường chéo dấu sắc.
- $dc\_huyen[i-j]=0$ : không có quân hậu trên đường chéo dấu huyền.



Hình 3.6. Minh họa quan hệ giữa các chỉ số  $i$  và  $j$  trên 2 đường chéo

- Điều kiện ô (i, j) an toàn:  $h[i]=dc\_sac[i+j]=dc\_huyen[i-j+7]=0$
- Điều kiện ô (i, j) không an toàn (bị chiếu):  $h[i]=dc\_sac[i+j]=dc\_huyen[i-j+7]=1$

- Để trực quan, ta xét ví dụ cụ thể sau: Đầu tiên ta “thử” ở cột 0. Trên cột 0 ta đặt quân hậu tại ô (0,0). Với cột 1 ta thử đặt quân hậu tại ô (0,1): trạng thái này không an toàn; ta thử đặt nó tại ô (1,1): cũng không an toàn; ta thử tiếp đặt nó tại ô (2,1). Tương tự ta thử đặt các con hậu ở các cột 2,3,... cho đến khi đủ 8 quân hậu mà chúng không ăn lẫn nhau (*Vết cạn*).

- *Quay lui* lại quân hậu trên cột 1, thử đặt nó tại ô (3,1), rồi tiếp tục đặt thử các quân hậu ở các cột 2,3,... Khi đã thử xong đến ô (7,1) lui lại thử đặt nó tại ô (1, 0). Cứ tiếp tục như vậy cho tới hết. Với các phân tích trên ta có cài đặt khá gọn sau cho bài toán 8 quân hậu với chiến lược quay lui:

#### e. Cài đặt bài toán 8 hậu

```
#include <stdio.h>
#include <conio.h>
int b[8][8], h[8], dc_sac[15], dc_huyen[15];
int dem=0;
void Out()
{
    dem++;
    printf("Loi giai thu %d:\n\n", dem);
    for(int i=0; i<8; i++)
    {
        for (int j=0; j<8; j++)
            printf("%2c ", b[i][j]==0?'.':'H');
        printf("\n");
    }
    getch();
}
void Try(int x)           //x: số hiệu cột của quân hậu đang thử
{
    for(int i=0; i<8; i++)
        if(h[i]==0 && dc_sac[x+i]==0 && dc_huyen
            [x-i+7]==0) //Nếu ô (i,x) an toàn
```

```

{ b[i][x]=1;           // thì đặt quân hậu vào ô (i, x)
  h[i]=dc_sac[x+i]=dc_huyen[x-i+7]=1; //hàng i, đg
                                         chéo x+i & đg chéo x-i không an toàn
  if (x==7)             // nếu đã thử hết 8 quân hậu (chỉ số hậu
                         đánh từ 0,1...)
    Out();              // thì xuất kết quả
  else
    Try(x+1);          //Ngược lại thử tiếp và quay lui chuyển tới
                         ô (i+1,x), muốn thế tái lập lại trạng thái đầu cho ô (i,x)
    b[i][x]=0;          h[i]=dc_sac[x+i]=dc_huyen[x-
                         i+7]=0;      //và tái lập trạng thái đầu cho các đường (hàng i,
                         }                  //chéo sắc / và chéo huyền )
}
int main()
{ clrscr(); int i,j;
  for (i=0; i<8; i++)
    for(j=0; j<8; j++) b[i][j]=0;
    for(i=0; i<8; i++) h[i]=0;
    for(i=0; i<15; i++) dc_sac[i]=dc_huyen[i]=0;
    Try(0);
  return 1;
}

```

Bài toán 8 quân hậu có 92 nghiệm, tuy nhiên nếu không kể các kết quả “đôi xứng ảnh gương” và các kết quả do xoay bàn cờ đi một góc nhất định thì bài toán đó chỉ có đúng **12 nghiệm cơ sở**.

### 3.3.3. Tổng quát hóa bài toán tám hậu

Ngày nay các nhà toán học đã tổng quát hóa bài toán trên đây là tìm cách đặt  $n$  quân hậu trên bàn cờ vua kích cỡ  $n \times n$  sao cho chúng không ăn được nhau. Cho tới lúc quyển sách này đang được soạn thì người ta đã tìm được 2680 lời giải cho trường hợp 11 quân hậu.

*Chiến lược thiết kế thuật toán vẫn là backtracking.* Ở mức tổng quát, ý tưởng chủ đạo để giải bài này như sau. Người ta dùng mảng  $\text{solution}[1..n]$  để lưu các lời giải; với  $\text{solution}[i]=j$  có nghĩa là quân hậu ở hàng  $i$  được *đặt vào cột  $j$* .

Cũng giống như trường hợp 8 quân hậu, các ô  $(i,j)$  thuộc đường chéo dấu sắc (/) có tổng  $i+j=\text{const}=\{2, 2n\}$ . Các đường chéo này được đánh số từ  $1 \rightarrow 2n-1$ . Nghĩa là các ô trên đường chéo / thứ nhất có tổng  $i+j=2$ , các ô trên đường chéo / thứ  $k$  có  $i+j=k+1$ . Dùng mảng kiểu Boolean  $\text{Ok\_plus}[1..2n-1]$  để kiểm tra xem một quân hậu nào đó đã ở trên đường chéo / thứ  $k$  chưa? Cụ thể  $\text{Ok\_plus}[1..2n-1]=1$  nếu đã có quân hậu chiếm giữ trên đường chéo / thứ  $k$ . Tương tự các ô trên đường chéo huyền \ có hiệu  $i-j=\text{const}=\{1-n, n-1\}$ . Chỉ số của các đường chéo \ được đánh số từ 1 đến  $2n-1$ . (đường chéo \ có  $i-j=1-n$  ứng với chỉ số 1; đường chéo \ có  $i-j=n-1$  ứng với chỉ số  $2n-1$ ). Khi đó đường chéo \ thứ  $k$  có hiệu số  $i-j=k-n$ , ta dùng mảng  $\text{ok\_minus}[1..2n-1]$  để đánh dấu trạng thái của đường chéo này.

Thuật toán này chúng ta cố gắng đặt quân hậu ở hàng  $i$  vào cột nào đó, bắt đầu từ dòng thứ nhất. Nếu ở dòng  $i$  ta đặt quân hậu vào cột  $j$  thì quân hậu này sẽ không chế được tất cả các ô thuộc cột  $j$ , đường chéo sắc / thứ  $i+j-1$ , đường chéo huyền \ thứ  $i-j+n$ . Nếu có thể đặt quân hậu ở hàng thứ  $i$  và  $i=n$  thì ta có một lời giải. Nếu đặt được và  $i < n$  thì ta tiếp tục thử tiếp theo là đặt quân hậu vào hàng thứ  $i+1$ . Nếu không được, ta quay lại nhắc quân hậu khỏi hàng thứ  $i-1$  và tìm phương án tiếp theo của hàng  $i-1$ .

Dưới đây là mã giả của thuật toán này (trường hợp tổng quát):

```
Try_row(i)
{
    for (j=1; j<n; j++)
        if (!ok_row[i] && !ok_col[j] && !ok_plus[i+j-1]
            && !ok_minus[[i-j+n]])
        {
            solution[i]=j;
            ok_col[j]=1;
            ok_plus[i+j-1]=1;
            ok_minus[[i-j+n]]=1;
            if (i<n)
```

```

        Try_row(i+1);
    else printf("Lời giải", solution[i]);
        ok_row[i] =0;
        ok_col[j]=0;
        ok_plus[i+j-1]=0;
        ok_minus[i-j+n]=0;
    }
}

```

Hàm tìm tất cả các lời giải bài toán n quân hậu chỉ gồm một lời gọi Try\_row(1)

```

N_queen(n)
{
    Try_row(1);
}

```

### 3.3.4. Bài toán Mã đi tuần (Ví dụ kinh điển 2 minh họa cho chiến lược quay lui)

*a. Nội dung bài toán:* Trên bàn cờ vua trống kích thước  $8 \times 8$ , quân mã đặt ở một ô tùy ý. Hãy tìm cách di chuyển nó theo luật cờ vua để con mã đi qua *mỗi ô đúng một lần* và *kết thúc* (hành trình của nó) *tại ô xuất phát*. (hành trình như thế gọi là *hành trình đóng*).

Thực chất của bài toán mã đi tuần là *tìm chu trình Hamilton* trong một ví dụ cụ thể trên đồ thị vô hướng (xem môn “Lý thuyết đồ thị”) biểu diễn bàn cờ vua  $8 \times 8$ .

Có rất nhiều lời giải cho bài toán trên, tính đến thời điểm này thì trên toàn thế giới đã có  $26.534.728.821.064$  lời giải trong đó con mã có thể kết thúc hành trình của nó tại chính ô xuất phát.

Vào năm 1823, lần đầu tiên H.C.Warnsdorff công bố thuật toán đầy đủ về bài toán mã đi tuần và nó được mang tên ông: thuật toán Warnsdorff. Lý thuyết ngắn gọn về bài toán này nằm trong định lý Schwenk

#### b. Định lý Schwenk

Cho bàn cờ vua kích thước  $m \times n$  ( $m \leq n$ ). Không tồn tại một hành trình đóng nào của con mã nếu một trong 3 điều kiện dưới đây xảy ra:

- 1)  $m$  và  $n$  đều lẻ
- 2)  $m=1, 2$  hoặc  $4$ ;  $n \neq 1$
- 3)  $m=3$  và  $n=4, 6$  hoặc  $8$

- *Điều kiện 1*

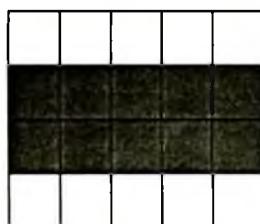
Dễ dàng chứng minh được khi điều kiện 1 xảy ra thì không có hành trình đóng nào của con mã. Trên bàn cờ vua, các ô đen-trắng xen kẽ nhau, vì thế một con mã đi từ ô tùy ý nào đó sẽ đến một ô khác màu. Vì  $m, n$  đều lẻ nên số các ô đen-trắng trên bàn cờ là khác nhau. Ví dụ bàn cờ  $5 \times 5$  thì có 13 ô đen và 12 ô trắng.

Một hành trình *đóng* của con mã phải có *số ô đen và ô trắng bằng nhau*. Và tổng số ô trên mọi hành trình *đóng* là *số chẵn*. Vì vậy một hành trình *đóng* không thể đi qua *mỗi ô đúng một lần* khi số các ô trên bàn cờ là *số lẻ*.

- *Điều kiện 2*

Dễ dàng thấy rằng khi  $m=1, 2, 4$  thì không thể có hành trình đóng của con mã với  $n \neq 1$ . Vì khi đó con mã không thể đi qua mọi ô (trừ bàn cờ kích thước  $1 \times 1$ ). Cũng có thể thấy rằng bàn cờ  $4 \times n$  (với  $n \neq 1$ ) cũng không có hành trình đóng của con mã.

Ta giả sử ngược lại có một hành trình đóng của mã trên bàn cờ  $4 \times n$ . Ta ký hiệu  $A_1$  là tập các ô đen,  $B_1$  là tập các ô trắng trên bàn cờ này. Theo luật cờ vua, con mã luôn di chuyển liên tiếp giữa 2 tập  $A_1$  và  $B_1$  (và ngược lại).



*Hình 3.7. Mô tả điều kiện 2*

Xét hình 3.7, trên hình này ta gọi  $A_2$  là tập các ô trắng,  $B_2$  là tập các ô đen. Các tập này có số ô bằng nhau. Rõ ràng từ một ô thuộc  $A_2$  con mã chỉ có thể nhảy sang một ô đen thuộc  $B_2$ , ngoài ra vì con mã phải đi qua tất cả các ô nên ngược lại nếu con mã đang đứng ở một ô thuộc  $B_2$  thì ở bước tiếp theo nó phải nhảy tới một ô thuộc  $A_2$ , (Nếu không thì trên hành trình đóng của con mã phải có 2 ô liên tiếp trong  $B_2$ , điều này không thể được).

Chúng ta tiếp tục lập luận để thấy mâu thuẫn trong giả thiết!

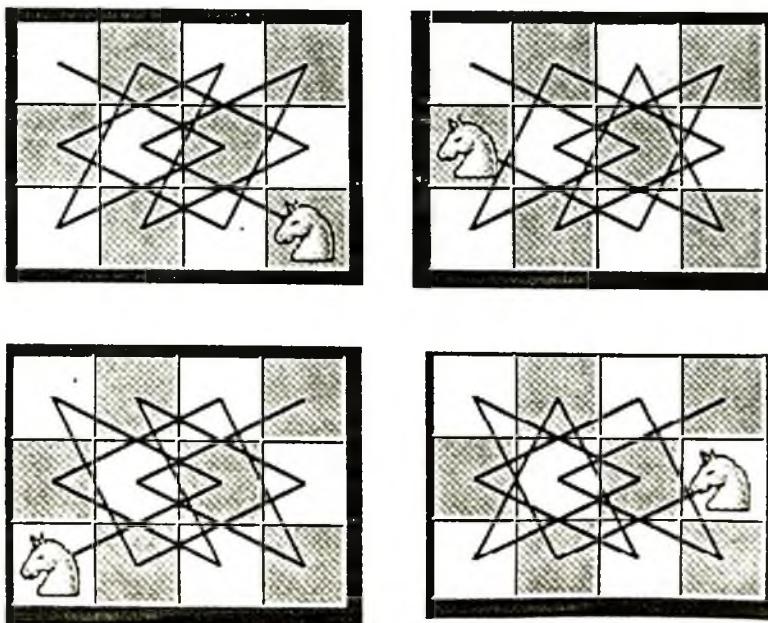
Vì hành trình của con mã là đóng nên chúng ta có thể chọn ô bất kỳ nào đó làm ô thứ nhất.

- 1- Chọn ô thứ nhất  $\in A_1 \cap A_2$
  - 2- Khi đó ô thứ hai phải  $\in B_1 \cap B_2$
  - 3- Ô thứ ba  $\in A_1 \cap A_2$
  - 4- Ô thứ tư  $\in B_1 \cap B_2$
- .....

Như vậy hành trình này không chứa các ô  $\in$  tập  $A_1 \cap B_2$  và tập  $B_1 \cap A_2$  nghĩa là không chứa tất cả các ô trên bàn cờ.

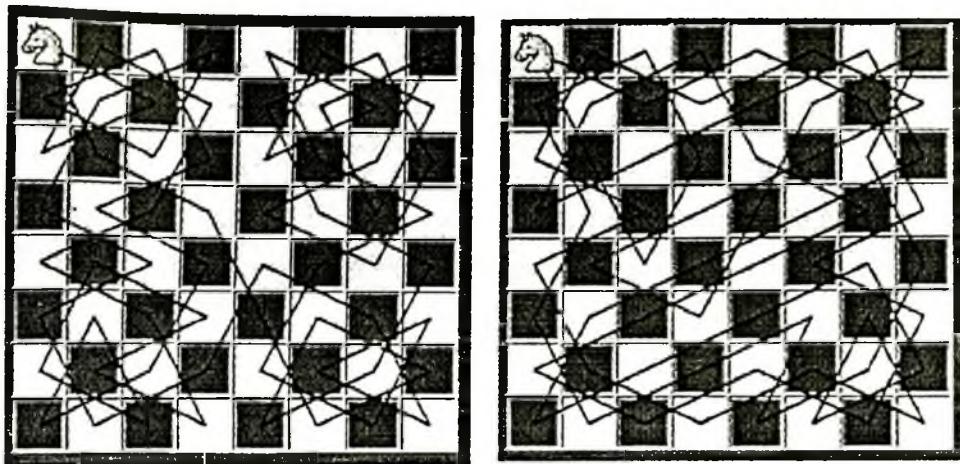
#### - Điều kiện 3

Điều kiện này được chứng minh cho từng trường hợp cụ thể. Nếu không yêu cầu hành trình đóng, thì điều kiện 3 xảy ra vẫn có thể có hành trình mở. Với bàn cờ  $3 \times 4$  chúng ta có 4 hành trình mở sau (hình 3.8).



Hình 3.8. Bàn cờ  $3 \times 4$  với 4 lời giải (4 hành trình mở của mã)

Hình 3.9 minh họa 2 trong số **rất nhiều** hành trình đóng của con mã trên bàn cờ  $8 \times 8$



Hình 3.9. Hai hành trình đóng của mã trên bàn cờ 8x8

### c. Cách giải bài toán mã đi tuần

Có nhiều cách giải bài toán này, ở đây ta dùng cách giải của H.C Warnsdorff kết hợp với đệ quy quay lui. Nguyên tắc trong cách giải của ông là quân cờ luôn phải được di chuyển tới một trong các vị trí mà từ đó nó có thể **đi qua số ít nhất** các vị trí chưa được đi qua. Nếu bàn cờ là một lưới các ô vuông có kích thước  $8 \times 8$  được biểu diễn bởi mảng  $a[i][j]$  và giả sử vị trí hiện tại của nó là  $a[i][j]$  thì vị trí kế tiếp có thể là một trong các vị trí sau:

$a[i-2][j-1]; a[i-1][j-2]; a[i+1][j-2]; a[i+2][j-1]; a[i+2][j+1]; a[i+1][j+2]; a[i-1][j+2]; a[i-2][j+1].$

Theo định lý Schwenk, nếu  $n$  chẵn (khoảng từ  $6 \rightarrow 16$ ) và trong trường hợp tồn tại đường đi của mã thì thuật toán H.C Warnsdorff là tối ưu: với mọi vị trí xuất phát, thời gian bắt đầu chạy chương trình tới lúc tìm ra một nghiệm là ít hơn 1 giây.

Tuy nhiên nếu  $n$  lẻ mà lại không có đường đi thì do phải duyệt hết mọi khả năng nên thời gian chạy chương trình là rất lâu (Có xét ưu tiên như phương pháp của H.C Warnsdorff hay xét theo phương pháp thứ tự truyền thống thì cũng như vậy. Khi test trên máy mà các trường trang bị cho sinh viên thực hành, thường sẽ mất rất nhiều thời gian. Khi đó, cách duy nhất phải làm để thoát khỏi việc này là gõ CTRL-BREAK!!! Dưới đây là một trong rất nhiều code hữu hiệu của bài toán:

**e. Cài đặt bài toán mā di tuần trên C++**

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
int p[8]={-2,-2,-1,-1,1,1,2,2};
int q[8]={-1,1,-2,2,-2,2,-1,1};
int d=0, x,y,m,n,a[50][50];
void inkq()
{ cout<<endl<<"Cach di thu:<<"+d<<": "<<endl;
  for(int i=0; i<m; i++)
  { for(int j=0; j<n; j++)
    printf("%3d", a[i][j]);
    cout<<endl;
  }
}
void tim(int x, int y, int k)
{
  if(k>m*n) inkq();
  else
    for (int i=0; i<m; i++)
    { int x1=x+p[i], y1=y+q[i];
      if(x1>=0 && x1<m && y1>=0 && y1<n)
        if (a[x1][y1]==0)
          { a[x1][y1]=k;
            tim(x1,y1,k+1);
            a[x1][y1]=0;
          }
    }
}
```

```

int main()
{ clrscr();
  cout<<"Nhập kích thước bàn cờ:"; cin>>m>>n;
  cout<<"Nhập vị trí xuất phát:"; cin>>x>>y;
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      a[i][j]=0;
  a[x][y]=1;
  tim(x,y,2);
  if (d==0)
    cout<<"Không có cách đi nào thỏa mãn yêu
          cầu!";
  else
    cout<<"Có tất cả "<<d<<" cách đi thỏa mãn
          yêu cầu!";
  inkq();
  getch();
  return 0;
}

```

*Lưu ý:* Code trên đã thử trên máy có cấu hình: Bộ nhớ vật lý 1 GB; bộ nhớ ảo 2GB với dữ liệu nhập vào: bàn cờ 8x8, tọa độ xuất phát của mã: hàng 3, cột 4. Sau 5 phút nhận được lời giải thứ 511!

### 3.4. THUẬT TOÁN THAM ĂN (GREEDY ALGORITHM - GA)

#### 3.4.1. Các thành phần (components) và hai đặc trưng của GA

Đây là một chiến lược giải quyết một bài toán theo kiểu tìm kiếm lựa chọn tối ưu cục bộ (Optimal Local) tại mỗi bước với hy vọng tìm được tối ưu toàn cục (Optimal Global).

Nhìn chung, thuật toán tham ăn có 5 thành phần:

- + Tập các ứng viên (Candidate Set) để từ đó kiến tạo lời giải.
- + Một hàm lựa chọn để lựa chọn các ứng viên tốt nhất bổ sung vào tập CS.

- + Một hàm khả thi (feasibility) để quyết định nếu một ứng viên có thể được dùng để kiến tạo lời giải.
- + Một hàm mục tiêu, xác định giá trị của lời giải hoặc một lời giải chưa hoàn chỉnh.
- + Một hàm đánh giá, chỉ ra khi nào ta tìm ra một lời giải hoàn chỉnh.

Có hai đặc trưng của GA:

#### *Tính chất lựa chọn tham ăn*

Tại mỗi bước lựa chọn, thuật toán này chọn “miếng ngon nhất” được xác định bởi hàm chọn “ngon nhất” có thể là giá trị max, hoặc min, hoặc theo một nghĩa nào khác tùy vào bài toán cụ thể) và nếu có thể “ăn” được (*có thể làm nghiệm bài toán*) thì nó “ăn” luôn, nếu không nó sẽ bỏ đi và *không bao giờ xem xét lại*.

Nói một cách khác, ở thời điểm hiện tại, ta lựa chọn giải pháp nào được cho là tốt nhất và sau đó giải bài toán con sinh ra ứng với sự lựa chọn vừa rồi. Lựa chọn của GA có thể phụ thuộc vào các lựa chọn trước đó, song không phụ thuộc vào một lựa chọn bất kỳ nào trong tương lai hay không phụ thuộc vào lời giải của bất kỳ bài toán con nào. GA tiến triển theo kiểu thực hiện các lựa chọn theo một vòng lặp đồng thời thu nhỏ bài toán đã cho về bài toán con nhỏ hơn.

Đó là điểm khác biệt với thuật toán Quy hoạch động (QHĐ). Thuật toán QHĐ *duyệt hết tất cả các trạng thái* và luôn đảm bảo tìm thấy lời giải. Với QHĐ, tại mỗi bước, việc đưa ra quyết định dựa trên các quyết định của bước trước và có thể xét lại hướng đi của bước trước tiến tới lời giải hoặc có thể xét lại quyết định cũ (điều này không xảy ra với thuật toán tham ăn!)

#### *Đặc trưng cấu trúc con tối ưu (Optimal SubStructure)*

Đặc trưng này là điểm duy nhất giống nhau của hai thuật toán GA và QHĐ.

*Một bài toán được gọi là có “cấu trúc con tối ưu” nếu một lời giải tối ưu của bài toán con chứa lời giải của bài toán lớn hơn.*

Đối với nhiều bài toán, chiến lược tham ăn không cho được lời giải optimal global (do đặc trưng “tham ăn” của thuật toán này mà nó không rà soát hết tất cả các trường hợp). GA *bám chặt lấy một lựa chọn quá sớm và không xét lại các quyết định cũ* nên *khó mà có thể cho được lời giải optimal global!* Tuy nhiên trong một số trường hợp cụ thể, GA vẫn có thể đưa ra lời giải optimal global và vì nó chạy nhanh hơn QHĐ nên nó vẫn được dùng cho lớp hẹp các bài toán loại đó.

Những ví dụ về áp dụng của GA là một vài bài toán trong “Lý thuyết đồ thị” như: Thuật toán Prim, thuật toán Kruscal tìm cây khung ngắn nhất (Minimal Spanning Tree), thuật toán Dijkstra (tìm đường ngắn nhất từ một đỉnh tùy ý đến các đỉnh còn lại trên đồ thị) và thuật toán tìm cây Huffman tối ưu.

### 3.4.2. Một ứng dụng của GA cho một bài toán cụ thể

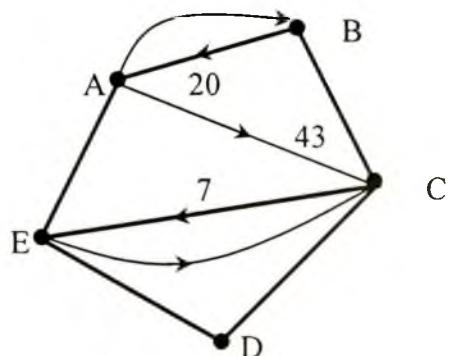
*Bài toán người du lịch (Tên gọi khác: bài toán người giao hàng - Travelling Salesman)*

a. *Phát biểu bài toán:* Có N thành phố được ký hiệu bởi các ký tự A, B, C,... (hoặc a, b, c...).

Một khách du lịch xuất phát từ một thành phố bất kỳ trong số các thành phố ấy và thăm các thành phố còn lại, mỗi thành phố chỉ thăm đúng 1 lần rồi trở về nơi xuất phát. Biết rằng  $C(i, j) > 0$  ( $\forall i \neq j$ ) là chi phí đi từ thành phố i đến thành phố j;  $C(i, i) = 0$ . Có thể  $C(i, j) \neq C(j, i)$ . Hãy tìm một hành trình đóng (chu trình) cho khách du lịch sao cho tổng chi phí là ít nhất. Lời giải sẽ kiểm thử (Test) trên một ví dụ cụ thể của chi phí du lịch  $C(i, j)$  dưới đây. Chi phí (Cost) du lịch  $C(i, j)$  được biểu diễn bởi ma trận trọng số (Trọng số trên đồ thị này là chi phí để đi từ i đến j).

b. *Tổ chức dữ liệu vào:* Chúng ta cần nhớ ghi số đỉnh (biểu diễn các thành phố) ở phía trên dòng đầu tiên của ma trận trọng số (Xem ma trận trọng số và đồ thị bên dưới).

5	
A	0 48 43 54 31
B	20 0 30 63 22
C	29 64 0 4 7
D	6 19 2 0 8
E	1 28 9 18 0



Hình 3.10. Đồ thị trọng số chưa vẽ hết các cạnh biểu diễn mạng các thành phố ứng với ma trận trọng số ở bên trái

#### c. Cài đặt trên C++

```
#include<iostream.h>
#include<fstream.h>
```

```

#include<stdio.h>
#include<conio.h>
int n, a[25][25], b[25], k; //5 là số đỉnh của đồ thị và cũng
//là kích thước mỗi chiều của ma trận trọng số//
int nhap(char ten[], int &n)
{
    ifstream iFile;
    iFile.open(ten);
    while (iFile>>n)
    {
        for(int i=0; i<n; i++)
        {
            for(int j=0; j<n; j++)
            {
                iFile>>a[i][j]; //Đọc dữ liệu từ file vào mảng
            }
        }
    }
    iFile.close();
    return 1; //Hàm nhập trả về 1 khi đọc thành công
}
int check(int x, int y[], int s)
{
    for(int i=0; i<s; i++)
    {
        if(x==y[i]) return 0;
    }
    return 1;
}
char toChar(int n)
{
    return 'A'+n;
}

```

```
}

int tim_va_xuat(int n, int id)
{
    int min;
    int b[100];
    int dem=0, tam, s=0;
    b[0]=k; //Lộ trình xuất phát từ thành phố k tùy ý
    while (dem<n)
    {
        min=10000; //Cứ chọn thành phố có chi phí nhỏ nhất
                     //là đến (từ th_phố hiện tại)
        for(int i=0; i<n; i++)
            if(a[id][i]<min && a[id][i]>0 &&
check(i,b,n)==1)//Điều kiện kiểm tra lộ trình thỏa mãn yêu cầu
            {
                min=a[id][i]; //Lưu chi phí nhỏ nhất
                tam=i; //Và đánh dấu địa điểm có chi phí nhỏ nhất
            }
        if(dem<n-1)
            s=s+min; //Rồi tính tổng chi phí cho lộ trình vừa thực hiện
        b[dem+1]=tam;
        id=tam;
        dem++; //Tiếp tục thăm thành phố khác
    }
    s=s+a[id][k]; //Tính tổng khi kết thúc lộ trình khép kín
    b[n]=k; //Trở về địa điểm xuất phát
    cout<<"\t Lo trinh dong kin cua khach du
          lich:";

    for(int i=0; i<n; i++)
        cout<<toChar(b[i])<<"->";
    cout<<toChar(b[n]);
```

```

        cout<<"\n\n";
        cout<<" Voi tong chi phi it nhat=" <<s;
        cout<<"\n\n";
        return 0;
    }

int main()
{
    char c; clrscr();
    nhap("D:\\\\TC\\\\BIN\\\\DTA_INPU.TXT", n); //Trong máy của
                                                //người dùng thư mục TC để ở ổ đĩa D
    cout<<"\t Ma tran trong so:" << endl; //Và tệp dữ liệu
                                                //vào có tên DTA_INPU.TXT
    for(int i=0; i<n; i++)
    {
        cout<<"\t" << toChar(i) << ":" ;
        for(int j=0; j<n; j++)
        cout<< " " << a[i][j];
        cout<<"\n\n";
    }
    cout<<"Cho biet thanh pho xuat phat cua khach
du lich:";

    cin>>c; //Khi nhập tên thành phố xuất phát, bạn gõ
              //chữ cái hoa
    if(c<='Z') k=c-65; //hoặc thường đều được vì đã có đoạn code
    //bên trái chuyển
    else k=c-97; // hoa → thường và ngược lại!
    cout<<"\n";
    tim_va_xuat(n, k);
    getch();
    return 0;
}

```

*Kết quả chạy chương trình khi bạn nhập thành phố xuất phát là A:*

Lộ trình đóng kín của khách du lịch: A → E → C → D → B → A

Với tổng chi phí ít nhất là: 83

Bạn có thể tự kiểm tra kết quả này trên giấy nháp bởi cách làm sau:

- Hãy quan sát ma trận trọng số (hoặc quan sát đồ thị hình 3.15) rồi ghi như dưới:

$$\begin{array}{ccccccc} - & A \rightarrow E \rightarrow C \rightarrow D \rightarrow B \rightarrow A \\ & \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \\ 31 & + & 9 & + & 4 & + & 6 & + & 20 = 83 \end{array}$$

### 3.5. THUẬT TOÁN QUY HOẠCH ĐỘNG

#### 3.5.1. Mở đầu

Với nhiều bài toán thực tế, phương châm “chia để trị” thường được dùng để thiết kế thuật toán giải chúng. Trong chiến lược QHD phương châm ấy được nâng lên mức độ cao hơn: khi không biết chắc cần giải quyết bài toán con nào thì người ta giải quyết tất cả các bài toán con đó và lưu những lời giải ấy (của các bài toán con) nhằm sau này sử dụng lại chúng theo một cách kết hợp nào đấy để giải quyết bài toán tổng quát hơn.

Như vậy, giống đê quy, QHD cũng dùng phương châm: “chia để trị” nhưng *khác về bản chất* ở chỗ: “quy trình” của QHD đi từ dưới lên (**Bottom-Up**). Từ cách giải các bài toán con rồi tìm cách kết hợp nghiệm của chúng để có lời giải bài toán lớn hơn) trong khi đê quy lại thiết kế thuật toán theo “quy trình” từ trên xuống (**Top-Down**).

Thuật ngữ “quy hoạch” ở đây ngũ ý nói đến quá trình đưa bài toán ban đầu về dạng nào đó để có thể áp dụng chiến lược nói trên để giải.

Một số khó khăn khi dùng chiến lược quy hoạch động để giải quyết một số bài toán:

- Không phải lúc nào sự kết hợp lời giải các bài toán con cũng cho lời giải chấp nhận được của bài toán lớn.

- Số lượng các bài toán con cần giải là rất lớn và vì thế số lượng lời giải của chúng rất lớn gây khó khăn cho việc lưu trữ. Cho đến nay chưa có ai có thể đưa ra được một lý thuyết tổng quát để xác định chính xác những bài toán nào có thể giải quyết hiệu quả bằng QHD. Cũng có những bài toán khá đơn giản khiến cho việc sử dụng QHD để giải quyết nó kém hiệu quả hơn so với việc dùng các thuật toán khác như đê quy, đê quy quay lui...

Quy hoạch động (QHĐ) thường được dùng để giải quyết các bài toán tối ưu. Song nó vẫn có thể dùng để giải những bài toán không có tính tối ưu, miễn là chúng ta biết cách “nhận diện” những bài toán đó giải được bằng phương pháp QHĐ.

### 3.5.2. Nhận diện các bài toán không tối ưu hóa có thể giải được bằng QHĐ

Muốn dùng được QHĐ để giải các bài toán phi tối ưu hóa thì chúng phải có tính chất: “các bài toán con phủ chồng”.

Chúng ta sẽ làm rõ khái niệm “các bài toán con phủ chồng”. Một thuật toán đệ quy cho bài toán ban đầu sẽ giải quyết lặp lại các bài toán con tương tự thay vì luôn phải giải quyết các bài toán con mới phát sinh. Trường hợp này thuật toán đệ quy gọi nhiều lần các bài toán con cùng loại, khi đó ta nói rằng bài toán đã cho có “các bài toán con phủ chồng”.

*Ví dụ 3.11:* Một minh chứng “kinh điển” và đơn giản nhất cho việc nhận diện các bài toán phi tối ưu mà áp dụng được QHĐ là bài toán xác định phần tử thứ n trong dãy Fibonacci. Dãy này định nghĩa như sau:

$$F_n = \begin{cases} 0 & \text{Nếu } n=0 \\ 1 & \text{Nếu } n=1 \\ F_{n-1}+F_{n-2} & \text{Nếu } n>1 \end{cases}$$

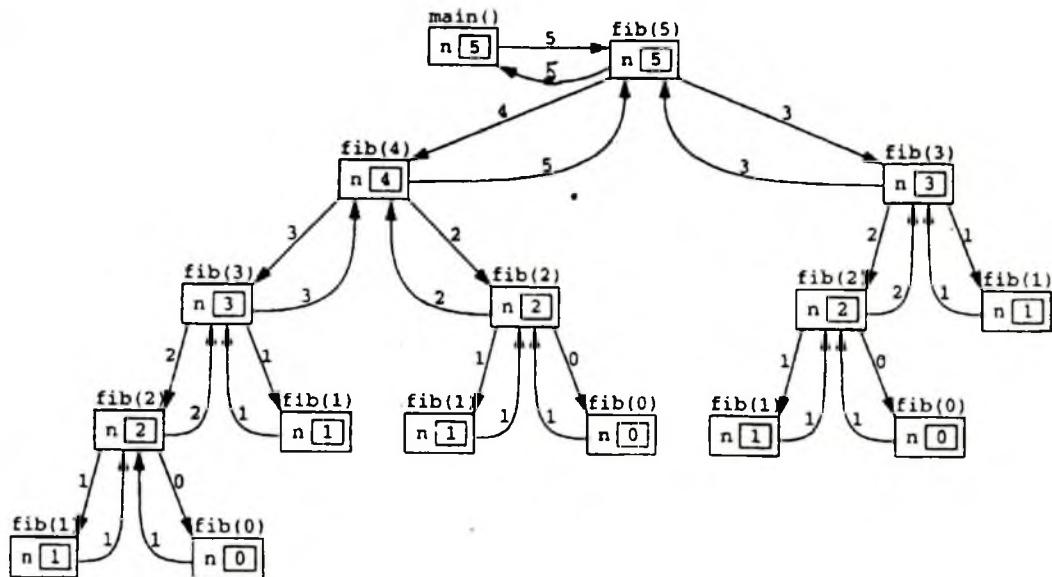
Hệ thức trên đây có bản chất toán học là công thức *truy hồi*, một công cụ luôn được dùng trong QHĐ.

Cách giải đệ quy của bài toán này khá quen thuộc với nhiều độc giả, tuy nhiên chúng tôi cũng nhắc lại hàm tính  $F_n$  rất đơn giản dưới đây:

```
long Fibo(int n)
{
    if (n<2)    return n;           // Trường hợp cơ sở
    return (Fibo(n-1)+Fibo(n-2)); // Đệ quy
}
```

Chẳng hạn chúng ta cần tính  $F(5)$ , sơ đồ sau cho thấy rõ cơ chế làm việc của hàm thiết kế trên đây, mỗi hàm Fibo() thực chất là một bài toán con. Sơ đồ khá trực quan, vì vậy ta dễ nhận thấy: đầu tiên hàm main() sẽ truyền dữ liệu 5 cho hàm Fibo(5). Để tính Fibo(5), bài toán con Fibo(5) gọi bài toán con Fibo(4) và bài toán con Fibo(3). Rồi bài toán con Fibo(4) gọi bài toán con Fibo(3) và bài

toán con Fibo(2). Đến lượt bài toán con Fibo(3) gọi bài toán con Fibo(2) và bài toán con Fibo(1) v.v... Như ở trên đã trình bày, đó là tính chất phủ chòng.



Hình 3.11. Minh họa các hàm phủ chòng khi tính giá trị  $F(n)$   
của dãy Fibonacci nhờ kỹ thuật đệ quy

Từ sơ đồ hình 3.11 ta thấy khi tính Fibo(5) đã phát sinh cây các lời gọi hàm. Các hàm cùng giá trị được gọi nhiều lần (làm tăng thời gian chạy chương trình!):

1. Fibo(5)
2. Fibo(4)+Fibo(3)
3. (Fibo(3)+Fibo(2))+(Fibo(2)+Fibo(1))
4. (((Fibo(2)+Fibo(1))+(Fibo(1)+Fibo(0)))+((Fibo(1)+Fibo(0))+Fibo(1)))
5. ((((Fibo(1)+Fibo(0))+Fibo(1))+Fibo(1)+Fibo(0)))+((Fibo(1)+Fibo(0))+Fibo(1))

Theo cách trên, nếu tính Fibo( $n$ ) với  $n$  cực lớn thì rất tốn thời gian và rất tốn bộ nhớ! Trong trường này, dùng QHĐ sẽ hiệu quả hơn!

Vì bài toán (không tối ưu) trên có các bài toán con phủ chòng nên áp dụng được QHĐ để giải nó. Muốn thế ta dùng một mảng  $A[]$  để lưu kết quả các bài toán con. Khi đó mỗi bài toán con chỉ phải tính một lần (mã giả):

1.  $A[0]=0;$
2.  $A[1]=1;$
3. Cho chỉ số  $i$  chạy từ 2 đến  $n$  tính  $A[i]=A[i-1]+A[i-2]$  //công thức truy hồi

### 3.5.3. Dùng QHĐ cho các bài toán tối ưu hóa

Với các bài toán tối ưu hóa, việc áp dụng QHĐ tỏ ra có hiệu quả và khá thuận tiện trong lập trình. Dưới đây là một vài bài toán tối ưu hóa thực tế.

#### a. Bài toán kinh điển “Xếp ba lô”

- *Nội dung bài toán:* Gia đình tôi đi du lịch với một ba lô đựng đồ có sức chứa một khối lượng tối đa là  $W$ . Chúng tôi muốn chất vào ba lô  $n$  đồ vật  $x_i$  với  $i=\{1, 2, \dots, n\}$  gồm:  $x_1, x_2, \dots, x_n$ . Mỗi đồ vật  $x_i$  có khối lượng  $w_i$  và giá trị  $p_j$ . Tìm một chiến lược để có thể chất vào ba lô nhiều đồ vật nhất (*đạt giá trị lớn nhất*) với *hạn chế khối lượng* là  $W$ .

- Các dạng cơ bản của bài toán “Xếp ba lô”: Bài toán xếp ba lô có các dạng sau:

- + Bài toán xếp ba lô 0/1.
- + Bài toán xếp ba lô phân số.
- + Bài toán xếp ba lô bị chặn
- + Bài toán xếp ba lô không bị chặn

Ta lần lượt nêu các diễn đạt toán học và cách giải chúng nhờ QHĐ.

##### a1. Bài toán xếp ba lô dạng 0/1

Đây là bài toán hạn chế số đồ vật mỗi loại là “không được chọn”  $\Leftrightarrow 0$  và “được chọn”  $\Leftrightarrow 1$ . Các điều kiện của bài toán này được diễn đạt bởi các hệ thức toán học sau:

$$\text{sum}_p = \sum_{j=1}^n p_j x_j \leftarrow \text{cực đại hóa tổng giá trị với hạn chế dưới đây:}$$

$$\text{sum}_w = \sum_{j=1}^n w_j x_j < W \text{ với } x_j=0 \text{ or } x_j=1; j=1, 2, \dots, n$$

*a<sub>2</sub>. Bài toán xếp ba lô bị chặn*

Bài toán này được diễn đạt bởi các công thức sau:

$$\text{sum}_p = \sum_{j=1}^n p_j x_j \leftarrow \text{cực đại hóa với các hạn chế dưới đây:}$$

$$\text{sum}_w = \sum_{j=1}^n w_j x_j < W \text{ với } 0 \leq x_j \leq b; j=1, 2, \dots, n$$

*a<sub>3</sub>. Bài toán xếp ba lô không bị chặn*

Bài toán này không có một hạn chế nào về số lượng đồ vật mỗi loại. Một dạng đặc biệt của bài toán này thường được quan tâm hơn có các tính chất sau:

- + Là một bài toán quyết định
- + Là một bài toán 0/1
- + Với mỗi loại đồ vật, chi phí là  $W=C$  ( $C$  là một hằng cho trước nào đó)

*Dạng đặc biệt trên đây* của Bài toán xếp ba lô không bị chặn tương đương với bài toán số học sau: *Cho trước một tập các số nguyên, có tồn tại chăng một tập con các số nguyên có tổng bằng C.*

Do vậy có lúc trường hợp đặc biệt này còn được gọi là bài toán “*tổng của các tập con*” (Subset Sum Problem - SSP).

Bài toán xếp ba lô thường được giải bằng QHD. Tuy nhiên đến nay chưa có thuật toán thời gian đa thức cho bài toán tổng quát. (*Thời gian đa thức (hay độ phức tạp đa thức) là thời gian thực thi thuật toán tỷ lệ thuận với lũy thừa bậc k của cỡ dữ liệu vào, tức là O(x<sup>k</sup>) (chú giải của tác giả)*).

Cả bài toán xếp ba lô tổng quát và các bài toán SSP đều là các bài toán NP - khó trị (*Các bài toán NP là các bài có độ phức tạp không đa thức tức là O(bài\_toán\_NP) ≠ O(x<sup>k</sup>) (chú thích của tác giả)*).

Các bài toán SSP còn có ứng dụng trong mật mã với khoá công khai, ví dụ mã Merkle - Hellman.

Phiên bản bài toán quyết định (một bài toán được gọi là *bài toán quyết định* nếu việc giải nó chỉ phải trả lời “Có” hoặc “Không”) của bài toán xếp ba lô mô tả ở trên là NP\_đầy đủ (Về NP\_đầy đủ xem phụ lục 1) và về thực chất là một trong 21 bài toán NP\_đầy đủ của Karp.

*a<sub>4</sub>. Bài toán xếp ba lô dạng phân số*

Ở bài toán này, mỗi loại đồ vật có thể chọn một phần nào đấy của nó (ví dụ mỗi cái bánh mì dài có thể cắt làm ba; mỗi kg bơ có thể chia bốn; mỗi hộp phomat có thể chia đôi...).

Bài toán xếp ba lô có thể giải trong thời gian tựa đa thức bằng QHĐ.

*b. Áp dụng QHĐ để giải bài toán xếp ba lô*

*Ý tưởng:*

Để xác định, chúng ta giải bài toán “Xếp ba lô không bị chặn” bằng QHĐ. Thuật toán này cũng khá đơn giản: Một cách tổng quát, ta ký hiệu hàm  $F[i][j]$  là giá trị max có thể đạt được khi xếp i đồ vật với giới hạn khối lượng là j. Khi đó:

- + Không chọn đồ vật nào thì hiển nhiên  $F[0, j] = 0$ ;
- + Công thức truy hồi:  $F[i,j]=F[i-1][j]$  nếu  $w_i > j$  ( $w_i$  – khối lượng đồ vật thứ i, j: là nguồn của tổng khối lượng).
- + Nếu chọn đồ vật thứ i + Điều kiện [Được giá trị lớn hơn] thì:

$$F[i,j]=F[i-1][j-Khối\_lượng[i]]+Giá\_trị[i]; (*)$$

Chúng ta cần làm cực đại giá trị biểu thức bên phải hệ thức (\*).

*Dưới đây là cài đặt trên C++ của bài toán Xếp ba lô nói trên.*

*c. Cài đặt trên C++*

```
#include <iostream>
#include <fstream>
using namespace std;
struct packed
{
    int W,V;
};
void Input(int &n, int &M, packed *&A)
{
    ifstream fi("BAG.INP");
    fi>>n>>M;
    A=new packed[n+1];
}
```

```
for (int i=1; i<=n; i++)
{
    fi>>A[i].W>>A[i].V;
    fi.get();
}
fi.close();
}
void Optimize(int n, int M, packed A[], int **&F)
{
    F=new int*[n+1];
    for (int i=0; i<=n; i++)
        F[i] = new int [M+1];
    memset(F[0], 0, M);
    for (int i=1; i<=n; i++)
        for (int j=0; j<=M; j++)
        {
            if (A[i].W<=j && A[i].V+F[i-1][j-A[i].W]>F[i-1][j])
                F[i][j]=F[i-1][j-A[i].W] + A[i].V;
            else
                F[i][j]=F[i-1][j];
        }
    }
void Output(int **F, packed A[], int n, int M)
{
    ofstream fo("BAG.OUT");
    fo<<F[n][M]<<endl;
    for (;n>0;n--)
        if (F[n][M] != F[n-1][M])
        {
            fo<<n<<endl;
            M-=A[n].W;
        }
    fo.close();
}
```

```

}

int main()
{
    int n,M;
    packed *A;
    int **F;
    Input(n,M,A);
    Optimize(n,M,A,F);
    Output(F,A,n,M);
    delete []A;
    for (int i=0;i<=n;i++) delete []F[i];
    delete []F;
}

```

### 3.6. THUẬT TOÁN NHÁNH VÀ CẬN

#### 3.6.1. Ý tưởng chủ đạo của chiến lược nhánh và cận

Chiến lược **nhánh và cận** (Branch and Bound Method - BB) là một cải tiến của chiến lược quay lui. Bản chất của BB là *tìm kiếm theo chiều sâu* của cây trạng thái có quay lui. Chỉ khác một điểm duy nhất là khi tới trạng thái  $a_k$  (nút  $a_k$ ) mà chi phí trên đường tìm kiếm  $P(a_1, a_2, \dots, a_{k-1}, a_k) \geq P_{\min}$  ( $P_{\min}$ : chi phí nhỏ nhất của tuyến tìm kiếm tốt nhất) thì ta “chặt” bỏ tất cả *các nhánh đi từ  $a_k$  xuống các nút con của nó* và quay trở lại nút cha của  $a_k$  là  $a_{k-1}$ . Khởi trị cho  $P_{\min}$  có thể là  $\infty$  hoặc là một trị rất lớn nào đấy so với trị *lớn nhất* của dữ liệu cho trong bài toán (chẳng hạn trên đồ thị tìm kiếm, trọng số lớn nhất là 200 thì bạn có thể chọn  $P_{\min} = 65536$ ).

Chiến lược **BB** thường được dùng để giải bài toán tối ưu.

#### 3.6.2. Tóm tắt bản chất của chiến lược BB

- + Sử dụng quay lui nhưng tại mỗi bước làm thêm việc đánh giá giá trị phương án hiện có.

- + Nếu đó là phương án tối ưu hoặc có hy vọng trở thành phương án tối ưu (tức là tốt hơn phương án hiện tại) thì cập nhật lại phương án tối ưu hoặc đi tiếp theo hướng ấy.

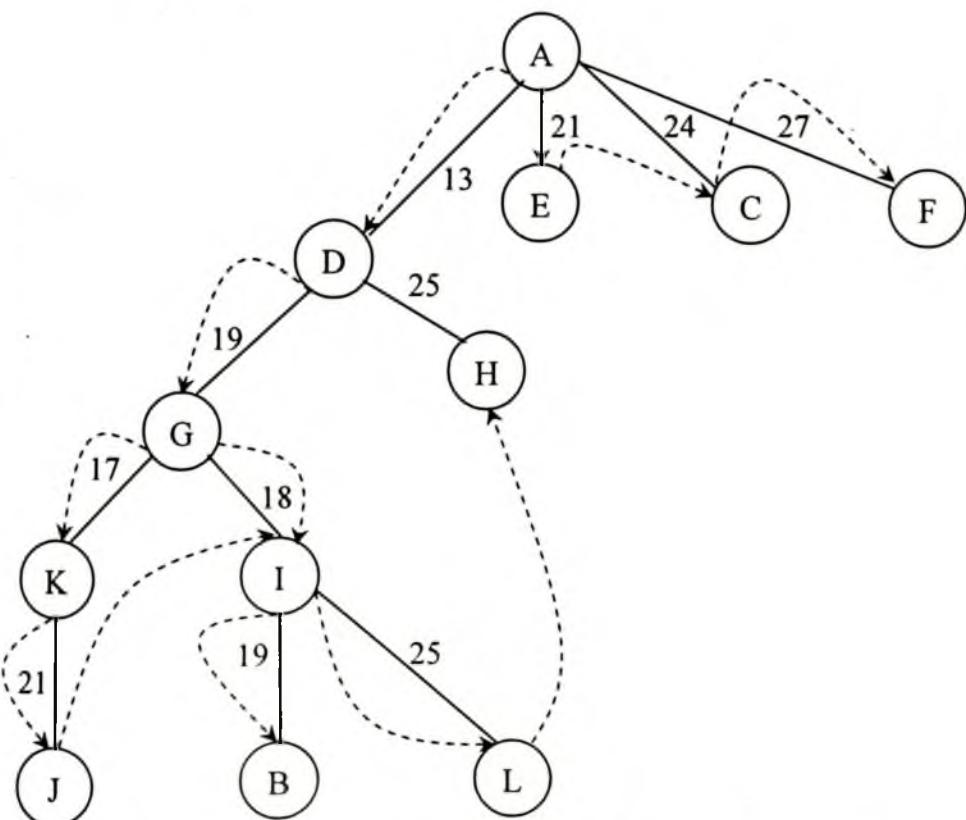
- + Trong trường hợp ngược lại thì bỏ qua hướng đang xét.

### 3.6.3. Mã giả tổng quát diễn đạt thuật toán

```

Try(i, P) //Sinh trạng thái thứ i thuộc cây tìm kiếm với chi phí
            hiện thời P
{ for (mỗi v thuộc tập các khả năng trạng thái của
      nghiệm xi)
  {if( v là chấp nhận được)
    Cp = P + Chi phí tìm nghiệm khi chấp nhận v;
    if (P tốt hơn Toptimize) //Tốt hơn chi phí tốt nhất
      xi = v ;               hiện có Toptimize
      if(xi là trạng thái kết thúc);
        Toptimize = P;         //Cập nhật chi phí tốt nhất
    else
      Try(i+1, P); //xét trạng thái tiếp theo với chi phí hiện thời T
  }
}

```



Hình 3.12. Tìm kiếm nhờ chiến lược nhánh và cận

*Ví dụ:* Chúng ta xét không gian trạng thái trong hình 3.12 với đích là B. Cơ chế hoạt động của phương pháp nhánh\_cận trên trường hợp này như sau:

Xuất phát từ đỉnh A, ta phát triển tới các đỉnh con C, D, E và F với các giá tương ứng là  $P(C) = 24$ ,  $P(D) = 13$ ,  $P(E) = 21$ ,  $P(F) = 27$ . Trong số này D là tốt nhất, phát triển D, sinh ra các đỉnh con H và G,  $P(H) = 25$ ,  $P(G) = 19$ . Đi xuống phát triển K, sinh ra các đỉnh con là K và I,  $P(K) = 17$ ,  $P(I) = 18$ . Đi xuống phát triển K sinh ra đỉnh J với  $P(J) = 21$ . Khi đi xuống J ta tìm được đường đi tối ưu hiện thời với độ dài 21. Từ J quay lên I, rồi từ L quay lên cha nó là H. Từ G đi xuống I,  $P(I) = 18$  nhỏ hơn độ dài đường đi tối ưu hiện thời (là 21). Phát triển I sinh ra các con B và L,  $P(B) = 19$ ,  $P(L) = 25$ . Khi đi xuống đỉnh B, vì đỉnh B là đích chúng ta tìm được đường đi mới với giá là 19 nhỏ hơn độ dài đường đi tối ưu tạm thời cũ (21). Vậy độ dài đường đi tối ưu tạm thời hiện tại là 19. Bây giờ từ B ta lại quay lên các đỉnh còn lại chưa được phát triển. Song các đỉnh này đều có giá lớn hơn 19, do đó không có đỉnh nào được phát triển nữa. Như vậy, ta tìm được đường đi tối ưu với độ dài 19 (Goal).

*Ví dụ:* Dùng lại bài toán “Người du lịch” ở mục 3.4.2/a (chỉ thay đổi cách đánh số các thành phố cho tiện)

Có n thành phố được đánh số từ 1 đến n. Một khách du lịch xuất phát từ một thành phố bất kỳ trong số các thành phố ấy và thăm các thành phố còn lại, mỗi thành phố chỉ thăm đúng 1 lần rồi trở về nơi xuất phát. Biết rằng  $C(i, j) > 0$  ( $\forall i \neq j$ ) là chi phí đi từ thành phố i đến thành phố j ;  $C(i, i)=0$ . Có thể  $C(i, j) \neq C(j, i)$ . Hãy tìm một hành trình đóng (chu trình) cho khách du lịch sao cho tổng chi phí là ít nhất. Lời giải sẽ kiểm thử (Test) trên một ví dụ cụ thể của chi phí du lịch  $C(i, j)$  dưới đây. Chi phí (Cost) du lịch  $C(i, j)$  được biểu diễn bởi ma trận trọng số. (Trọng số trên đồ thị này là chi phí để đi từ i đến j).

*Tổ chức dữ liệu:*

Biểu diễn mạng lưới giao thông giữa các thành phố là một đồ thị có trọng số  $G=\{V, E\}$ , V: tập các đỉnh, E: tập các cung.

- + Mỗi thành phố là một đỉnh của đồ thị (đánh số 1,...,n)
- + Mỗi đường đi giữa các thành phố là một cung nối giữa các đỉnh của đồ thị, có thể có hướng hoặc vô hướng, trên đó có ghi trọng số là chi phí đường đi.
- + Các cặp cạnh không có đường đi, trọng số là  $\infty$ .
- + Đường đi tìm được là dãy  $L=x_1, x_2, \dots, x_n$ ,  $x_1$  với  $x_i \in V$ ,  $(x_i, x_{i+1}) \in E$ , có tổng chi phí nhỏ nhất.

*Cơ chế hoạt động của thuật toán BB trên bài toán cụ thể này:*

- + Ký hiệu chi phí tốt nhất là BestP. Ban đầu BestP =  $\infty$ ;
- + Tại mỗi bước chọn  $x_i$ : Chi phí đường đi từ  $x_1$  đến  $x_{i-1}$  là C;
- + Với mỗi khả năng v, tính chi phí  $C_p = P + \text{chi phí từ } x_{i-1} \text{ tới } v$ ;
- Nếu  $C_p$  xấu hơn (lớn hơn) BestP hoặc không có khả năng nào chấp nhận được cho  $x_i$  thì lùi lại bước trước để xác định lại thành phần  $x_{i-1}$ ;
- Nếu  $C_p$  tốt hơn (nhỏ hơn) BestP thì chấp nhận  $x_i$  theo khả năng v. Tiếp tục xác định  $x_{i-1}, \dots$ ;
- + Đến khi gặp nghiệm ( $i=n+1$  &  $x_i=x_1$  thì trở về nơi xuất phát):
- Cập nhật đường đi tốt nhất hiện tại đã tìm được;
- Cập nhật giá trị BestP mới: BestP =  $C_p$ ;
- Kết thúc tìm kiếm nếu  $BestP = \infty \rightarrow$  không có đường đi hoặc đã tìm thấy nghiệm tối ưu.

#### *Tổ chức dữ liệu*

- Đánh chỉ số các đỉnh của đồ thị từ 1...n
- Biểu diễn đồ thị G bằng ma trận kè M = C[i, j] cỡ n x n; C[i, j] = p nếu có cạnh nối đỉnh i với đỉnh j; chi phí p =  $\infty$  nếu không có đường đi.
- Mảng: Datham[1...n] đánh dấu đỉnh i đã được thăm trên đường đi hay chưa?
- + Datham[i] = 1 nếu đỉnh i đã thăm; Datham[i] = 0 nếu ngược lại, đỉnh i chưa thăm (chưa có trên đường đi)
- + Khởi tạo: Datham[1...n] = 0

*Mã giả mô tả thuật toán nhánh\_cận cho bài toán “Người du lịch”*

Try(i, P) //Phát triển đến trạng thái thứ i của đồ thị với chi phí hiện thời P

```

{ for (v = 1..n)
    { if ((C[xi-1, v] < infinity) && (not Datham[v]))
        { Cp = P + C[xi-1, v]; //Tổng tích lũy chi phí đi từ xi-1
          đến v
            if (Cp < BestP) //Tốt hơn chi phí tốt nhất hiện có
                { xi = v; //Ghi nhớ trạng thái v

```

```

Datham[v] = 1; //v đã được thăm
if (i == n+1) && (xi == x1)
    BestP = Cp; //Cập nhật chi phí tốt nhất
else if (i<=n)
    Try(i+1,Cp); //Quay lui, phát triển trạng thái
                    tiếp theo
}
//với chi phí Cp
Datham[v] = 0; //v chưa được thăm
}
// if ngoài cùng của for
}
}

```

### BÀI TẬP CHƯƠNG 3

#### I. Thiết kế hàm đệ quy để giải các bài toán sau:

3.1. (*Hàm Ackermann*) Tính giá trị của Ackermann A(m, n) với  $0 \leq m \leq 3$  và  $0 \leq n \leq 8$ . Hàm này được định nghĩa như sau:

$$A(m,n) = \begin{cases} n+1 & \text{nếu } m=0; \leftarrow \text{basic case} \\ A(m-1,1) & \text{nếu } n=0 \text{ và } m>0 \leftarrow \text{basic case} \\ A(m-1,A(m, n-1)) & \text{nếu } m>0, n>0 \end{cases}$$

3.2. In lên màn hình tam giác Pascal chứa các hệ số của khai triển nhị thức  $(x+1)^n$ . Yêu cầu: xuất lên màn hình hai tam giác Pascal “vuông” và tam giác Pascal “đều” có cấu trúc dưới đây:

1 1		1 1
1 2 1		1 2 1
1 3 3 1		1 3 3 1
1 4 6 4 1		1 4 6 4 1
1 5 10 10 5 1		1 5 10 10 5 1
1 6 15 20 15 6 1		1 7 21 35 35 21 7 1
.....		.....

3.3. Tính  $n!!$ . Sau đó dùng hàm này để tính tổng:

$$S = 1!! - 2!! + 3!! - 4!! + \dots + (-1)^{k+1} k!! \text{ với } k < 1000$$

trong đó  $n!!$  được định nghĩa như sau:

$$n!! = \begin{cases} 1.3.5\dots(2k+1), & k=0,1,2\dots \\ 2.4.6\dots2k \end{cases}$$

3.4. Tính các tổng sau:

a.  $S = \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2n}$

b.  $S = 1+2^2+3^3+4^4+\dots+n^n$

c.  $S = \sqrt{3n + \sqrt{3(n-1) + \dots + \sqrt{9 + \sqrt{6 + \sqrt{3}}}}}$

d.  $S = \sqrt[n]{n} + \sqrt[n-1]{n-1} + \dots + \sqrt[i]{i} + \dots + \sqrt{2}$

e.  $S = \sqrt[n+1]{n + \sqrt[n]{n-1 + \sqrt[3]{2 + \sqrt[3]{1}}}}$  (có n dấu căn)

f.  $S = \sqrt{n + \sqrt{n-1 + \sqrt{n-2 + \dots + \sqrt{2 + \sqrt{1}}}}}$  (có n dấu căn)

g.  $S = \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1}}}}}}$

Có n dấu phân số

$$\cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + 1}}}$$

h.  $S = \sqrt[n+1]{n!} + \sqrt[n]{(n-1)!} + \dots + \sqrt[3]{2!} + \sqrt{1!}$  Có n dấu căn

i.  $S(n) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$  ( $x$  là số thực)

3.5. Cho lưới nhị phân sau, ta gọi các ô trong lưới này là liên thông 1 với nhau nếu chúng có chung cạnh và cùng chứa bit 1. Ở ví dụ dưới đây số miền liên thông 1 là 3.

1	0	1	0	1
1	1	1	0	1
0	1	0	0	0
0	1	1	0	1

Dữ liệu vào lưu trên file Dlieu.vao có cấu trúc như sau:

- + Dòng đầu là hai kích thước của lưới ô vuông  $m, n$ .
  - +  $m$  dòng tiếp theo ghi các số nhị phân ở các ô thuộc lưới.
- Lập trình đếm số miền liên thông 1 của lưới đã cho.

## II. Quy hoạch động: Xây dựng thuật toán và nếu có thể cài đặt nó cho các bài toán sau:

3.6. Cho xâu  $S \leq 1000$  ký tự, tìm Palindrome dài nhất là xâu con của  $S$ .

a. Palindrome là xâu đối xứng, nghĩa là xâu viết từ trái qua phải hay từ phải sang trái thì xâu ấy không thay đổi. Ví dụ: tát; ioi; radar...;

b. Xâu con của xâu đã cho  $S$  là dãy các ký tự liên tiếp thuộc  $S$  mà độ dài của nó không quá độ dài của  $S$ .

3.7. Chia một xâu cho trước  $S$  ( $L(S) \leq 1000$ ) thành **ít nhất** các palindrome.

3.8. Cho  $S$  là một xâu Palindrome có  $n$  chữ cái hoa, chữ cái thường phân biệt và các chữ số. Cho biết cần xóa đi ít nhất bao nhiêu ký tự từ  $S$  để thu được một Palindrome? Giả thiết rằng sau khi xóa các ký tự còn lại trong  $S$  sẽ xích lại gần nhau. (*Olympic Tin học quốc tế năm 2000, Bắc Kinh, trung Quốc*).

3.9. **Bài toán Cắm hoa:** Có  $n$  lọ hoa và  $k$  bó hoa được đánh số thứ tự từ nhỏ tới lớn. Cần cắm  $k$  bó hoa vào  $n$  lọ sao cho hoa có số thứ tự nhỏ phải đứng trước hoa có số thứ tự lớn. Giá trị thẩm mỹ khi cắm bó hoa  $i$  vào lọ  $j$  là  $v(i, j)$ . Hãy tìm một cách cắm hoa sao cho tổng giá trị thẩm mỹ là lớn nhất, biết rằng mỗi bó hoa chỉ được cắm vào một lọ và mỗi lọ chỉ cắm một bó hoa.

Dữ liệu vào cất trên text file với tên Camhoa.In gồm:

- Dòng 1: ghi 2 số n và k.

- k dòng tiếp theo mỗi dòng ghi n số nguyên dương với số ở hàng i cột j biểu thị giá trị thảm mỹ bó hoa i cắm vào lọ j.

Kết quả ghi lên text file với tên Camhoa.Ou gồm :

- Dòng đầu: In tổng giá trị thảm mỹ lớn nhất:

- k dòng tiếp theo mỗi dòng ghi 2 số i, j biểu thị bó hoa i cắm vào lọ j.

Một trường hợp cụ thể của bài toán cho ở hai file sau:

Camhoa.In	Camhoa.Ou
4 3	24
7 8 9 10	1 1
10 9 8 7	2 2
5 6 7 8	3 4

### III. Quay lui: Xây dựng thuật toán và nếu có thể cài đặt nó cho các bài toán sau:

3.10. Liệt kê tất cả các dãy nhị phân dài n bit.

3.11. Cho bộ chữ cái có 9 chữ cái khác nhau: a, b, c, d, e, f, g, h, i. Mỗi “từ” (word) lập nên từ (from) bộ chữ ấy gồm từ 2 đến 4 chữ cái, trong đó có ít nhất 1 nguyên âm. Đếm và liệt kê tất cả các từ khác nhau tạo ra từ bộ chữ cái đã cho.

3.12. Cho trước một tập gồm n số nguyên. Hãy liệt kê mọi cách phân hoạch tập này thành các tập con.

Ví dụ: S = {1, 2, 3} có thể phân hoạch thành {1}, {2}, {3}, {1, 3}, {2, 3}...

3.13. **Thuật toán Ghép cặp:** Có một lớp học sinh gồm N người (đánh số thứ tự từ 1 đến N và N <= 1000) đi cắm trại tham gia trò chơi như sau: mỗi em nam ghép cặp với một em nữ (và ngược lại mỗi em nữ ghép cặp với một em nam). Người ta dùng một tham số nguyên dương w để đánh giá độ ăn ý của các cặp ghép đó. Yêu cầu tìm những cặp ghép cho tổng độ ăn ý lớn nhất.

Dữ liệu vào cất trên text file với tên Ghepdoi.In gồm:

- Dòng 1 là số N.

- Các dòng tiếp theo mỗi dòng gồm 3 số x y w với ý nghĩa: x là số hiệu của nam sinh, y số hiệu của nữ sinh, w là độ ăn ý của 2 bạn đã ghép cặp với nhau.

Bt1.In	Bt1.Ou
4	4
1 2 2	1 3 1
1 3 1	4 2 3
4 2 3	
4 3 1	

Kết quả ghi lên text file với tên Ghepdoi.Ou gồm:

- Dòng đầu chứa một số nguyên dương là tổng độ ăn ý lớn nhất của các cặp ghép.

- Các dòng tiếp theo mỗi dòng ghi 3 số x, y, w.

Một ví dụ cụ thể cho ở phía phải các dòng trên đây!

### GỢI Ý HOẶC ĐÁP ÁN

3.1. Code của bài này là:

```
#include<iostream.h>
#include<conio.h>
int ackerm(int m, int n)
{ if(m==0) return n+1; //trường hợp cơ bản
  else if (n==0) return ackerm(m-1,1); //trường hợp cơ bản
  else return ackerm(m-1, ackerm(m,n-1)); //độ quy
}
int main()
{ clrscr();
  cout<<" CAC GIA TRI CUA HAM ACKERMANN VOI M=3 VA
N=8:\n\n";
  for (int m=0; m<=3; m++)
  {
```

```

for (int n=0; n<=8; n++)
{
    cout.width(5);
    cout<<ackerm(m, n);
}
cout<<endl;
}
getch();
}

```

3.4-i. Theo bài ra ta có:

$$S(n) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{n-2}}{(n-2)!} + \frac{x^{n-1}}{(n-1)!} + \frac{x^n}{n!}$$

$$S(n-1) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{n-2}}{(n-2)!} + \frac{x^{n-1}}{(n-1)!}$$

$$S(n-2) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{n-3}}{(n-3)!} + \frac{x^{n-2}}{(n-2)!}$$

- Tính hiệu  $S(n) - S(n-1)$  ta được:  $S(n) = S(n-1) + \frac{x^n}{n!}$  (\*)

- Tính hiệu  $S(n-1) - S(n-2)$  ta được:  $S(n-1) = S(n-2) + \frac{x^{n-1}}{(n-1)!}$

$$\rightarrow \frac{x}{n} S(n-1) = \frac{x}{n} S(n-2) + \frac{x^{n-1}}{(n-1)!} \frac{x}{n}$$

$$\rightarrow \frac{x}{n} S(n-1) = \frac{x}{n} S(n-2) + \frac{x^n}{n!}$$

$$\rightarrow \frac{x^n}{n!} = \frac{x}{n} S(n-1) - \frac{x}{n} S(n-2)$$

$$\rightarrow \frac{x^n}{n!} = \frac{x}{n} (S(n-1) - S(n-2)) (**)$$

Thay (\*\*) vào (\*) ta được công thức để thiết kế đệ quy:

$$S(n) = S(n-1) + \frac{x}{n} (S(n-1) - S(n-2))$$

3.7. Bài này khó hơn bài 3.6, thuật toán để giải nó như sau:

Gọi  $F[i]$  là số palindrome ít nhất mà đoạn  $1..j$  chia thành được. Ta có công thức:

$$F[i] = \max(F[j] + 1; \text{"}j < i \text{ thỏa mãn: đoạn } j+1..i \text{ palindrome}")$$

Ta có giả mã dưới đây:

```

F[0] := 0;
for(int i = 1; i<n; i++)
{
    for (int j = i-1; j>0; j--)
        if (đoạn j+1...i là palindrome)
            F[i] := max( F[i], F[j]+1 );
}

```

Hai vòng for lồng nhau mất  $O(N^2)$ , phần kiểm tra đoạn  $j+1...i$  có là xâu đối xứng hay không mất  $O(N)$ , vậy độ phức tạp thuật toán là  $O(N^3)$ . Sẽ không khả thi nếu  $N = 1000$ . Để giảm độ phức tạp thuật toán, ta sử dụng mảng  $L[i, j]$  có ý nghĩa tương tự như mảng  $F[i, j]$  ở Ví dụ 1, lập mảng  $L[i, j]$  mất  $N^2$ . Tổng cộng là  $O(N^2)$  với mỗi lần kiểm tra chỉ mất  $O(1)$ .

Nhưng đến đây lại nảy sinh vấn đề: mảng  $L[i, j]$  không thể lưu được dữ liệu khi  $N=1000$  vì bộ nhớ PC hiện nay có hạn. Một cách khắc phục “sự cố” này là dùng xử lý bit. Nhưng lại có cách đơn giản hơn là dùng hai mảng một chiều  $L[i]$  và  $C[i]$  có ý nghĩa:

\*  $L[i]$  là độ dài lớn nhất của palindrome có độ dài lẻ nhẵn  $s[i]$  làm tâm;

\*  $C[i]$  là độ dài lớn nhất của palindrome có độ dài chẵn nhẵn  $s[i+1]$  làm tâm;  $L[i]$  và  $C[i]$  có thể tách được theo cách 2 giải bài 1 (duyệt có đặt cận) mất  $O(N^2)$ . Phần kiểm tra ta có mã giả như sau:

```

int is_palindrome(int i, int j)
{

```

```

int dd = j-i+1;
if (dd)
    is_palindrome = (L[(i+j) / 2] >= n) ;
else is_palindrome := (C[(i+j) / 2] >= n)
}

```

Vậy thuật toán của ta bây giờ có ĐPT tính toán là  $O(N^2)$ , chi phí bộ nhớ là  $O(N)$ .

3.8. Ta dùng mảng 1\_chiều với quy hoạch động như sau:

- + Gọi  $p[i,j]$  là độ dài của **dãy con dài nhất** thu được khi giải bài toán với dữ liệu vào là xâu  $s[i,j]$ .
- + Khi đó  $p[1,n]$  là độ dài của **dãy con đối xứng dài nhất** thuộc dãy  $n$  ký tự  $s[1, n]$  và do đó số ký tự cần xóa bỏ khỏi xâu  $s[1, n]$  là:

$$n-p[1, n] \quad (\text{đáp số của bài toán})$$

- + Nếu  $i>j$  (chi số của vị trí đầu trái lớn hơn chi số của vị trí đầu phải của xâu) thì  $p[i, j]=0$

- + Nếu  $i=j$  (xâu có duy nhất một ký tự) thì  $p[i, j]=1$  (xâu có một phần tử độc nhất luôn là đối xứng!)

- + Nếu  $i<j$  và  $s[i]=s[j]$  thì  $p[i,j]=p[i+1, j-1]+2$ . Vì hai ký tự đầu, cuối xâu giống nhau nên chỉ cần xác định độ dài của xâu con đối xứng dài nhất thuộc đoạn giữa là  $s[i+1, j-1]$  rồi cộng với 2 (đó là hai các ký tự đầu, cuối xâu)

- + Nếu  $i<j$  và  $s[i] \neq s[j]$  tức là 2 ký tự đầu và cuối xâu con  $s[i, j]$  khác nhau thì ta xét 2 xâu con  $s[i, \dots, (j-1)]$  và  $s[(i+1, \dots, j)]$  và lấy độ dài của **dãy con đối xứng dài nhất** trong hai dãy đó làm kết quả:

$$p[i, j]=\text{Max}(p[i, j-1], p[i+1, j])$$

Đến đây bài toán quy về việc tính  $p[1, n]$ , mà để tính được  $p[1, n]$  chúng ta phải tính được  $p[i, j]$  với  $\forall i, j=1, \dots, n$ .

- + Để thuật toán gọn nhất có thể ta dùng mảng một chiều  $L[i]$  để lưu giữ độ dài  $p[i, j]$ . Nghĩa là tại từng bước cập nhật thứ  $j$  với mỗi  $i=(j-1)\dots 1$  ta gán  $a[i]=p[i,j]$  và mảng  $L[]$  được tính như sau:

- Nếu  $s[i] = s[j]$  thì:

$$\begin{array}{c} L[i]_j = L[i+1]_{j-1} + 2 \\ \downarrow \quad \searrow \\ L[i] \text{ tại bước } j \quad L[i+1] \text{ tại bước } j-1 \end{array}$$

- Nếu  $S[i] \neq s[j]$  thì :

$$L[i]_j = \text{Max}(L[i]_{j-1}, L[i+1]_j)$$

+ Nhưng khi tính  $L[i]$  từ cuối xâu trở lên tức là tính  $L[i]$  với  $i=n, n-1, \dots, 1$  thì  $L[i+1]$  sẽ bị ghi đè  $\rightarrow$  mất dữ liệu. Để tránh sự cố này chúng ta dùng hai biến tạm là  $temp1$  và  $temp2$

3.10. Ta có thuật toán sau:

Bước 1: Tạo một Stack chứa một dãy nhị phân. Đặt  $i=1$  (xét phần tử đầu tiên)

Bước 2: Lần lượt cho  $b[i]$  các giá trị từ 0 đến 1. Với mỗi giá trị của  $b[i]$  thực hiện:

- Nếu  $i < n$  thì tăng  $i$  lên một đơn vị (nghĩa là xét phần tử tiếp theo) và quay lại bước 2.

- Nếu  $i=n$  thì in dãy  $b$ .

Với  $n=3$  ta có quy trình liệt kê cho ở bảng sau:

Thứ tự	i	b[i]	Chú thích
1	1	0	
2	2	00	
3	3	000	In kết quả
4	3	001	In kết quả
5	2	01	Tăng j ở vòng for thứ 2
6	3	010	In kết quả
7	3	011	In kết quả
8	1	1	Tăng j ở vòng for thứ 1
9	2	10	
10	3	100	In kết quả
11	3	101	In kết quả
12	2	11	Tăng j ở vòng for thứ 2
13	3	110	In kết quả
14	3	111	In kết quả

3.12. Ta dùng quy nạp mô tả thuật toán phân hoạch tập s đã cho. Giả thiết quy nạp:

Tập  $S=\{1, 2, 3, \dots, n-1\}$  có các cách phân hoạch là  $\{S_1, S_2, S_3, \dots, S_k\}$ . Khi ấy tất cả các cách phân chia tập  $S=\{1, 2, 3, \dots, n\}$  là:

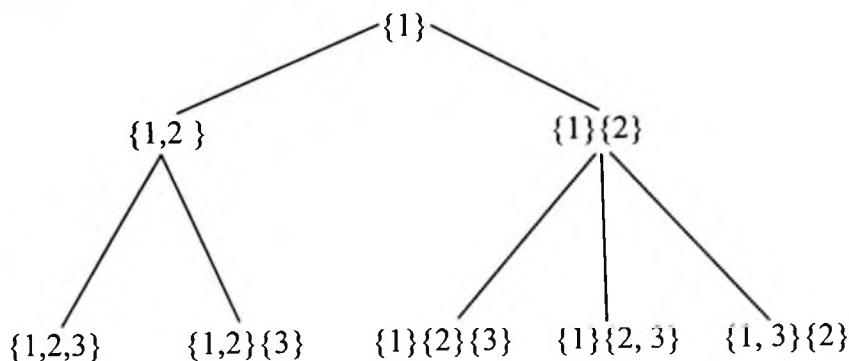
$$S_1 \cup \{n\}, S_2, S_3, \dots, S_k$$

$$S_1, S_2 \cup \{n\}, \dots, S_k, \dots$$

$$S_1, S_2, \dots, S_k \cup \{n\}$$

$$S_1, S_2, \dots, S_k, \{n\}$$

Nếu tập có 1 phần tử thì chỉ có 1 cách phân hoạch  $S=\{1\}$ . Để trực quan chúng ta mô tả cách phân hoạch tập  $S=\{1, 2, 3\}$  nhờ cấu trúc cây:



Cấu trúc cây mô tả thuật toán phân hoạch  $S=\{1, 2, 3\}$

## Chương 4

# CÁC KIỂU DỮ LIỆU TRÙU TƯỢNG VÀ BIẾN NHỚ ĐỘNG

*Triết lý đương đại về dữ liệu và sự kết hợp:*

**BIG DATA + BIG COMMUNITY = BIG RESULT**

(Dữ liệu lớn + Kết hợp lớn = Kết quả lớn)

### 4.1. CÁC KIẾN THỨC CƠ BẢN VỀ KIỂU DỮ LIỆU TRÙU TƯỢNG (KIỂU DỮ LIỆU ĐỘNG)

#### 4.1.1. Đặt vấn đề

Trong thực tế có khá nhiều bài toán cần quản lý và xử lý những đối tượng mà các kiểu dữ liệu đã có trước đây không phù hợp (chưa có giải pháp tối ưu) trước hai việc nêu trên. Để khắc phục “sự cố” này các “tác giả” của những ngôn ngữ lập trình đưa ra một loạt kiểu dữ liệu khác gọi là “các kiểu dữ liệu trùu tượng” (Abstract Data Type - ADTs). Các ADTs gồm:

- Danh sách.
- Các danh sách liên kết (đơn, kép, vòng...).
- Ngăn xếp (Stack).
- Hàng đợi (Queue; Hàng đợi thông thường, Hàng đợi ưu tiên...).

*Ví dụ 4.1.* Bất kỳ một cơ sở kinh doanh nào (nhà hàng, công ty, ngân hàng...) cũng cần phải quản lý nhân viên và các tác vụ tác động lên họ thường ngày: chấm công, thưởng, phạt, xét lên lương, sa thải, tuyển người mới... Các kiểu dữ liệu đã học như mảng (array), bản ghi (struct)... đều không dùng được cho bài toán quản lý nhân viên vừa nêu. Với bài toán này, kiểu dữ liệu phù hợp là danh sách liên kết đơn. Với ADTs này bạn có thể cập nhật mọi thứ cho bài toán quản lý nhân viên bất kỳ lúc nào!

Dưới đây là một ví dụ khác ít “cực đoan” hơn ví dụ 1, song cũng đủ để minh chứng tính bất lợi và có lợi giữa việc dùng các kiểu data đã học trước đây với việc dùng các ADTs như thế nào.

*Ví dụ 4.2.* Để quản lý điểm thi một môn nào đó của sinh viên (không quản lý tên hoặc phách, chỉ đơn thuần điểm thi thôi!) bạn có thể dùng mảng 1 chiều. Nhưng số sinh viên dự thi của các lớp khác nhau vào những thời điểm khác nhau luôn thay đổi và thay đổi bất thường. Giả sử có 5 lớp thi một môn nào đó, và lớp đông sinh viên nhất (số lượng trên giấy!) là 100, còn lại số sinh viên đều ít hơn, thậm chí có lớp ít hơn nhiều. Khi lập trình cho bài toán đơn giản này, bạn phải khai báo kích cỡ mảng là 100, mặc dù bạn biết chắc thực tế có lúc không dùng hết kích cỡ đó. Và khi đã khai báo (đăng ký) kích cỡ như vậy thì bạn phải để nguyên giá trị đó *trong suốt thời gian chạy chương trình*: bạn không thể giảm nó, cũng không thể hủy nó khi bạn đã không cần nó. Đây là một lãng phí lớn trong việc sử dụng bộ nhớ của máy tính.

Khi sử dụng các ADTs bạn không mặc phải “sự cố” dùng không gian nhớ lãng phí một cách vô lý như trên!

Các kiểu dữ liệu ta đã học và đã dùng trước đây (gồm các kiểu cơ sở: như kiểu ký tự, kiểu nguyên, kiểu thực và các kiểu có cấu trúc đơn giản như: mảng, tệp, bản ghi (struct)) gọi là các kiểu *dữ liệu tĩnh*. Nhược điểm chung của những kiểu dữ liệu tĩnh là:

- + Khai báo kích thước và cấu trúc từ đầu như thế nào thì phải dùng và để nguyên như vậy trong suốt thời gian chạy chương trình, nói cụ thể hơn: chúng ta không thay đổi được kích thước và cấu trúc của các kiểu dữ liệu tĩnh.

- + Như 2 ví dụ trên đã nêu, các kiểu dữ liệu tĩnh bắt lực trong việc mô tả một loạt các đối tượng của nhiều bài toán thực tế.

- + Ngoài ra các kiểu dữ liệu này bị “rào cản” 64 KB bộ nhớ của hệ điều hành. Nghĩa là nếu chúng ta cần không gian nhớ lớn hơn thì không được đáp ứng. Khi cần không gian nhớ lớn hơn bắt buộc phải dùng ADTs

#### 4.1.2. Định nghĩa kiểu dữ liệu trừu tượng

*Kiểu dữ liệu trừu tượng là một tập hợp các đối tượng mà những đối tượng ấy được mô tả bởi các kiểu dữ liệu tĩnh kèm theo (bắt buộc) một kiểu dữ liệu “đặc biệt” gọi là *kiểu con trỏ (pointer)* cùng với các thao tác (operations) trên chúng.*

Các ADTs còn gọi là các kiểu (hay cấu trúc) *dữ liệu động*.

*Ví dụ 4.3.* Dưới đây mô tả (định nghĩa) một kiểu ADTs là danh sách liên kết đơn (chúng ta sẽ nghiên cứu sau) để quản lý một vài thông tin về sinh viên.

```

typedef struct _____
{
    char ten[25];
    char MSSV[7];
    float DTB;
} SV;

typedef struct tagNode
{
    SV Info;
    struct tagNode *pNext; —→ Dữ liệu
                           kiểu con trỏ (gọi tắt là con trỏ)
} Node;

typedef struct tagList
{
    Node *pHead; —→ Dữ liệu kiểu con trỏ
    Node *pTail; —→ Dữ liệu kiểu con trỏ
} List;

```

Mô tả một kiểu dữ liệu trừu tượng (ADTs) là *danh sách liên kết đơn* dùng quản lý một số dữ liệu của sinh viên.

Các kiểu dữ liệu cơ sở trong C/C++

#### 4.1.3. Sự khác nhau giữa ADTs với cấu trúc dữ liệu

- Kiểu dữ liệu trừu tượng là một tập hợp các đối tượng mà những đối tượng ấy được mô tả bởi các kiểu dữ liệu tĩnh kèm theo (bắt buộc) một kiểu dữ liệu “đặc biệt” gọi là **kiểu con trỏ** cùng với các thao tác trên chúng.

- Còn cấu trúc dữ liệu là một **kết cấu** (construct) nằm sẵn trong một ngôn ngữ lập trình bậc cao, nó như là một kho dữ liệu.

#### 4.1.4. Kiểu dữ liệu con trỏ và biến thuộc kiểu con trỏ

##### a. Định nghĩa kiểu dữ liệu con trỏ (Gọi tắt là **kiểu con trỏ** và ký hiệu: $T_p$ )

Kiểu con trỏ  $T_p$  là kiểu dữ liệu **trỏ tới địa chỉ** các phần tử có kiểu  $T$  ( $T=\{V,O\}$  xem lại mục 1.2.1) và nó được định nghĩa là:

$$T_p = \{V_p, O_p\}$$

Trong đó

-  $V_p$  là tập các **địa chỉ** của các phần tử có kiểu  $T$  mà  $P$  **trỏ tới**.

-  $O_p$  là tập các thao tác **tác động lên địa chỉ** của phần tử có kiểu  $T$

**Kết luận:** kiểu con trỏ  $T_p$  là kiểu **chỉ dùng để lưu địa chỉ** của phần tử có kiểu  $T$  mà **không lưu dữ liệu** của nó.

**b. Biến thuộc kiểu con trỏ (Chúng ta sẽ gọi là biến con trỏ hoặc gọn nữa là con trỏ)**

#### - Biến tĩnh và biến động

Thích ứng với các dữ liệu kiểu tĩnh (char, int, long int, float, double, array, struct...) mà chúng ta đã nghiên cứu và dùng trước đây là **các biến tĩnh**. Chúng được gọi là các biến tĩnh vì:

- + Chúng được xác định trước một cách *tường minh* lúc khai báo biến, rồi chúng được dùng thông qua tên của chúng. *Thời gian sống của chúng là thời gian tồn tại của khỏi chương trình hoặc chương trình chưa khai báo các biến ấy.*
- + Không thể thay đổi kích thước hay cấu trúc của các biến tĩnh.
- + Cũng không thể hủy được biến tĩnh trong chương trình.

Các biến tĩnh có các đặc điểm cần nhớ và hiểu để sau này phân biệt với biến con trỏ (hoặc với biến động): Biến tĩnh vừa lưu nhớ kiểu vừa lưu nhớ dữ liệu của đối tượng mà ta xử lý trong chương trình, nó không lưu địa chỉ vùng nhớ bị dữ liệu chiếm dụng.

**Ví dụ 4.4.** Khi ta khai báo float dtb rồi gán dtb=7.25 thì biến dtb sẽ “nhớ” cả kiểu của nó và giá trị đã gán cho nó là 7.25.

Thích ứng với các dữ liệu động là các biến động (Dynamic variable): **Biến động là biến được tạo ra trong lúc chạy chương trình.** Điều ấy cũng có nghĩa là: biến động không được xác định (định nghĩa) trước và các **biến động không có tên**. Để truy cập các biến động phải dùng một biến “đặc biệt” gọi là **biến con trỏ (Pointer Variable)** được định nghĩa như sau:

#### - Định nghĩa biến con trỏ (Pointer Variable)

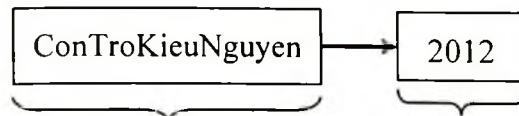
Biến con trỏ (hay con trỏ) P là biến chỉ lưu địa chỉ vùng nhớ của dữ liệu kiểu T mà không lưu dữ liệu kiểu T hoặc nó lưu giá trị NULL (rỗng, con trỏ P không trỏ tới đâu cả).

#### - Cú pháp khai báo biến con trỏ P

Có hai dạng cú pháp để khai báo biến trỏ P có kiểu:

+ Dạng thứ nhất: <kiểu dữ liệu> \*TênBiếnTrỏ;

**Ví dụ 4.5** int \*ConTroKieuNguyen; \*ConTroKieuNguyen;=2012;



Biến con trỏ, trỏ tới địa chỉ cát dữ liệu 2012

+ Dạng thứ 2: `typedef <kiểu dữ liệu> *TênBiếnTrỏ;`

*Ví dụ 4.6*      `typedef int * ConTroKieuNguyen;`  
`ConTroKieuNguyen P;`

**Khai báo biến trỏ không kiểu:** `void *TênBiếnTrỏ;`

- *Cấp phát vùng nhớ và giải phóng (hủy) vùng nhớ cho biến trỏ trong C (còn gọi là tạo một biến động)*

+ **Cấp phát vùng nhớ cho các dữ liệu kiểu cơ sở:** Trường hợp này dùng hàm `malloc()` với cú pháp tổng quát sau:

`(datatype*) malloc (kích thước vùng nhớ cần cấp phát); ← của C`

*Ví dụ 4.7* `int *P ;`

`P=(int*) malloc(200);` (cấp phát vùng nhớ 200 Bytes và gán địa chỉ vùng nhớ đó cho P, 200 bytes này để lưu 100 số nguyên (kiểu int))

Nếu muốn cấp phát vùng nhớ để lưu 125 số nguyên thì phải viết:

`P=(int*) malloc(125*sizeof(int))`

+ **Cấp phát vùng nhớ cho các dữ liệu có kiểu do người dùng tự định nghĩa:** Trường hợp này dùng hàm `calloc` với cú pháp:

`(DataType*) calloc(n, sizeof(đối_tượng)); ← của C`

Trong đó `DataType*` là kiểu con trỏ trỏ tới kiểu dữ liệu `DataType`, `n`: số lượng `đối_tượng` thuộc kiểu `DataType`, toán tử `sizeof()` quy định số Bytes cấp phát cho `đối_tượng`;

*Ví dụ 4.8.* Ta có một bản ghi (struct) quản lý 3 thông tin của sinh viên và sẽ cấp phát vùng nhớ cho bản ghi này:

```
struct sv
```

```
{ char hoten[30];
    int namsinh;
    char lop[15];
}
```

```
struct sv *p //Khai báo con trỏ sv
```

Lệnh cấp phát vùng nhớ để quản lý 60 sinh viên: `calloc(60, sizeof(struct sv))`

**- Toán tử cấp phát (tạo biến động) vùng nhớ của C++:** new với cú pháp rất đơn giản như sau: (Giả sử ta đã có biến trả P)

P=new <datatype>; // cấp phát vùng nhớ cho 1 phần tử

P=new <datatype>[n]; // cấp phát vùng nhớ cho n phần tử

*Ví dụ 4.9*

```
int      *sn=new int;
```

```
int *nhieu_sn=new int[205];
```

Nếu cấp phát vùng nhớ thành công thì biến con trả sẽ trả về địa chỉ đầu vùng nhớ được cấp phát, nếu thất bại biến trả sẽ trả về giá trị NULL.

**- Giải phóng vùng nhớ đã dùng**

+ Trong C: free(P) // hủy vùng nhớ do p trả tới

+ Trong C++: delete(P) // hủy vùng nhớ được cấp bởi new do P trả tới

**- Cấp lại vùng nhớ nhờ hàm realloc()**

Vùng nhớ được cấp lại bởi hàm realloc với cú pháp sau:

```
(datatype*) realloc(befo_p, new_size)
```

Trong đó, befo\_p là con trả trả tới vùng nhớ đã cấp phát trước đây, new\_size là kích thước mới cần cấp phát có thể lớn hơn hoặc nhỏ hơn vùng nhớ cấp phát cũ.

## 4.2. DANH SÁCH TUYẾN TÍNH TRÙU TƯỢNG

Danh sách tuyến tính trừu tượng (The Linear List ADT) - Gọi gọn là danh sách, viết tắt: DS.

Cho trước kiểu dữ liệu  $T=\{V, O\}$  (xem lại mục 1.2.1), danh sách  $T_L$  là tập các phần tử cùng kiểu T được xác định như sau:

$$T_L = \{V_L, O_L\}$$

Trong đó:

$V_L$ : Tập các phần tử có thứ tự *tuyến tính* cùng kiểu T. Ta nói “thứ tự tuyến tính” nghĩa là các phần tử của  $T_L$  được sắp tuyến tính: phần tử  $a_i$  đứng trước (kề từ trái sang) phần tử  $a_{i+1}$  với  $i=1, 2, 3, \dots, n$ .

Trong đó  $n$  là số phần tử thuộc danh sách  $T_L$  ( $n$  còn gọi là độ dài của  $T_L$ ). Chỉ số (index)  $i$  biểu thị cho vị trí của các phần tử  $\in T_L$ , nghĩa là muốn truy cập vào một phần tử tùy ý của danh sách ta truy cập theo chỉ số  $i$ .

$O_L$ : Tập các thao tác (operations) trên các phần tử của  $T_L$ . Các thao tác ấy là: Khởi tạo danh sách, tìm một phần tử trong danh sách, chèn thêm một phần tử vào (đầu, giữa, cuối) danh sách, hủy một phần tử thuộc danh sách, sắp xếp danh sách, xem (duyệt) danh sách v.v...

Danh sách hầu như không dùng cho các bài toán lớn ứng dụng trong thực tế, vì tính cập nhật thông tin cho các phần tử của danh sách là yếu và kém mềm dẻo. Người ta thường dùng các danh sách liên kết để cài đặt cho các ứng dụng thực tế phức tạp, do chúng có tính cập nhật dễ dàng và linh hoạt hơn danh sách.

### 4.3. DANH SÁCH LIÊN KẾT TRÙU TƯỢNG

Danh sách liên kết trùu tượng (The Single Linked List ADT) - Gọi gọn là danh sách liên kết đơn, viết tắt: DSLKD.

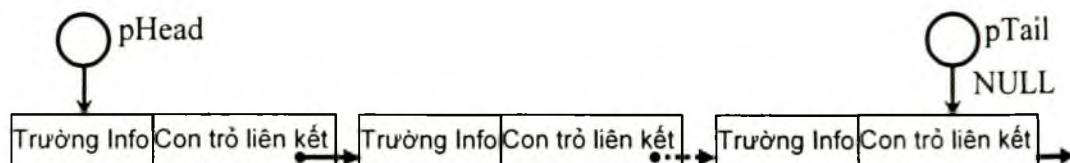
**4.3.1. Định nghĩa DSLKD:** Định nghĩa như mục 3.2.1, chỉ thêm hai kiến thức quan trọng:

- + Mỗi phần tử thuộc DSLKD được *liên kết (móc nối)* với phần tử đứng *kè liền sau nó* theo trình tự tuyến tính nhờ *con trỏ* trả tới *địa chỉ* của phần tử kế tiếp ấy.

- + Mỗi phần tử chỉ có *một* liên kết với một phần tử kè liền sau nó (vì vậy nó có tên gọi là Danh sách Liên kết *Đơn*).

**Kết luận:** DSLKD là danh sách mà một phần tử tùy ý liên kết với phần tử kè liền sau nó thông qua *địa chỉ* của phần tử này nhờ dùng (*một*) biến *con trỏ* trả tới *địa chỉ* của phần tử kế tiếp đó.

Sơ đồ dưới đây cho bạn hình ảnh trực quan về cấu trúc của DSLKD



Hình 4.1. Cấu trúc của danh sách liên kết đơn

- Các phần tử thuộc **DSLKd** còn gọi là Nút (Node). Nhìn sơ đồ ta thấy mỗi Node trong **DSLKd** có 2 trường:

- + Trường Info *cất giữ dữ liệu* kiểu *DataType* của đối tượng cần quản lý.
- + Trường Con trỏ Liên kết *lưu địa chỉ* của phần tử kế tiếp.

- pHead là con trỏ đầu **DSLKD** nó lưu địa chỉ của phần tử đầu tiên của danh sách này.
  - pTail là con trỏ cuối **DSLKD** nó lưu địa chỉ của phần tử cuối của danh sách này.
  - Giá trị **NULL** nghĩa là rỗng (con trỏ liên kết không trỏ tới đâu cả, tức *không có node kế tiếp*)

#### **4.3.2. Các thao tác cơ bản trên DSLKD**

#### a. Khai báo tổng quát cấu trúc DSLKD

`typedef struct` → Không nhất thiết phải có từ khóa `typedef`  
(xem ví dụ 4.10.a dưới đây)

{

```
    DataType_1    tp_1;           //tp:thành phần  
    DataType_2    tp_2;  
    . . . . .  
    DataType_k    tp_k;
```

} Bct\_1; —————> không nhất thiết phải có biến này (xem ví dụ 4.10.a dưới đây)

(Bct=Biến cấu trúc)

//Khai báo con trỏ liên kết:

```
typedef struct LL //LL:Linked List
```

```
Bct_1 Info; //Info: trường lưu các thông tin của đối  
           //tương cần quản lý
```

```
struct LL *pNext; //con trỏ pNext mόc nόi vόi phάn tύ (nút)  
//kέ tiέp (rō hօn):
```

```
} Bct_2; //pNext lưu địa chỉ của nút kế tiếp (với nút  
//hiện hành)
```

//Khai báo con trỏ trả về tới địa chỉ đầu (gọi tắt là con trỏ đầu) và con trỏ trả về tới địa chỉ cuối (gọi tắt là con trỏ cuối) danh sách:

```
typedef struct LKD //danh sách liên kết đơn
```

```
{ Bct_2 *pHead; //định nghĩa con trỏ đầu danh sách
```

```
Bct_2 *pTail; //định nghĩa con trỏ cuối danh sách
}Bct_3;
```

*Ví dụ 4.10*

- a. Trường hợp đơn giản nhất, khai báo một danh sách liên kết đơn chứa các số nguyên.

```
struct IntList
{ info;
  IntList *next;
};

struct Ds_sn
{
  IntList *pHead;
  IntList *Tail;
};
```

DSLKD lưu các số nguyên

- b. Trường hợp phức tạp hơn, khai báo danh sách liên kết đơn quản lý một số thông tin của sinh viên.

```
typedef struct
{ char ten[25];
  char masv[7];
  float DTB;
} SV

typedef struct Qlsv
{ SV Info; // trường Info lưu dữ liệu của các sinh viên
  struct Qlsv *pNext; // Con trỏ pNext lưu địa chỉ của Node
                      // (nút) kế tiếp
} dssv;

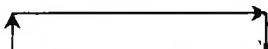
typedef struct ds
{ dssv *pHead; //Khai báo con trỏ đầu Ds
  dssv *pTail; //Khai báo con trỏ cuối Ds
} List;
```

(Độc giả chú ý những từ in đậm. Những bạn đọc mới tiếp cận với môn này hay nhầm lẫn khi khai báo một danh sách liên kết đơn để quản lý những đối tượng phức tạp).

### b. Khởi trị cho DSLKD

```
void Init(Bct_3 &L)      //Hàm khởi trị cho ds
{
    L.pHead=NULL;
    L.pTail=NULL;   { Ban đầu danh sách rỗng
}
}
```

*Ví dụ 4.11.* Khởi trị cho danh sách sinh viên (đã khai báo ở Ví dụ 4.10b) ban đầu rỗng (không có phần tử nào):



```
void Init(List &L) //hãy chú ý kiểu này (nó đã được khai báo ở ví dụ 4.10.b và là kiểu của tham số hình thức L thuộc hàm Init. Đầu & trước L: truy xuất địa chỉ của L.
```

```
{
    L.pHead=NULL;
    L.pTail=NULL;
}
```

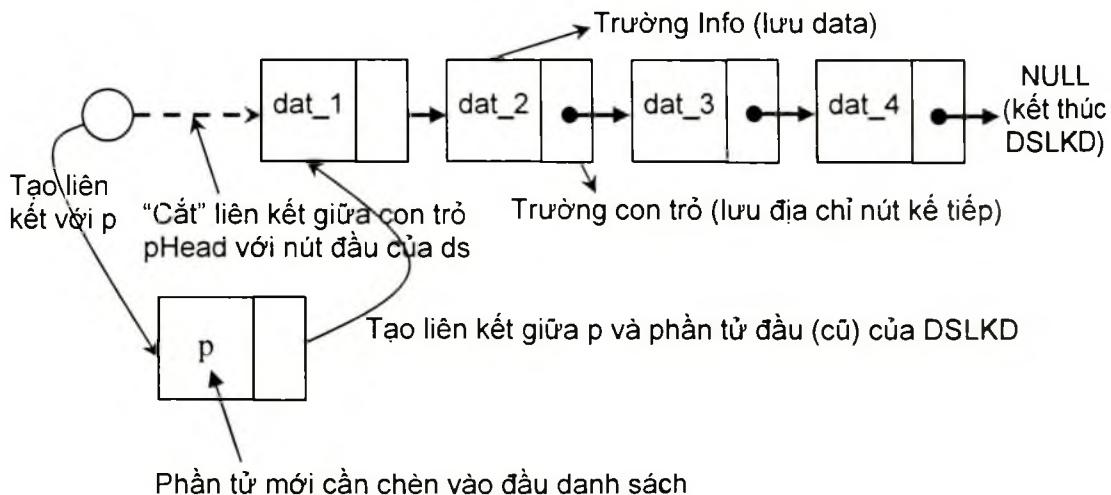
### c. Tạo nút mới cho DSLKD

```
Bct_2 *TaoNutMoi(Bct_1 x)
{
    Bct_2 *Newnode;
    Newnode = new Bct_2;           //Hàm cấp phát vùng nhớ cho biến động Newnode
    if (Newnode==NULL)
    {
        printf("khong du bo nho");
        exit(1);
    }
    Newnode->Info =x;//Nạp dữ liệu x vào trường Info của Newnode
    Newnode->pNext = NULL;       //Không có Node đứng ngay sau
    return Newnode;              //Hàm TaoNutMoi trả về địa chỉ của Newnode
}
```

*Ví dụ 4.12.* Trở lại danh sách quản lý sinh viên ở ví dụ 4.10b, ta xây dựng hàm tạo nút mới như sau:

```
dssv *TaoNutMoi (SV X)
{
    dssv *Newnode;
    Newnode = new dssv;           //Hàm cấp phát vùng nhớ cho
                                   biến động Newnode
    if (Newnode==NULL)
    {
        printf("khong du bo nho");
        exit(1);
    }
    Newnode->Info = x;          //Nạp dữ liệu x vào trường Info
                                 của Newnode
    Newnode->pNext = NULL;      //Chưa có Node dùng ngay sau
    return Newnode;             //Hàm TaoNutMoi trả về địa chỉ
                                của Newnode
}
```

*d. Chèn 1 nút mới vào đầu DSLKD:* Đề nắm được mô tả thao tác này trên C, trước hết cần xem sơ đồ hình 4.2 dưới đây:



*Hình 4.2. Mô tả thực chất của thao tác chèn phần tử mới vào đầu DSLKD với tình huống DSLKD không rỗng*

```

//Hàm mô tả thuật toán chèn phần tử mới vào đầu DLLKD
void InserHead(List &L, Bct_2 *p) // p: phần tử mới cần
                                         chèn vào đầu danh sách
{
    if (L.pHead==NULL) //nếu danh sách rỗng thì chèn thêm được
    {
        L.pHead = p;           //nạp phần tử mới p vào đầu danh sách
        L.pTail = L.pHead;     //cập nhật lại địa chỉ của con trỏ
                               //cuối danh sách
    }
    else
    {
        // ngược lại ta phải “cắt” mốc nối giữa
        // pHead với phần tử đầu danh
        //sách, rồi tạo mốc nối giữa phần tử mới
        // p với phần tử đầu (cũ) của
        p->pNext = L.pHead; //danh sách và cho con trỏ đầu ds pHead
        //trở tới đầu danh sách mới (tức trỏ tới p)
        L.pHead = p;          // (xem sơ đồ mô tả cơ chế làm việc
                               // của thao tác này ở trên)
    }
}

```

*Ví dụ 4.13.* Dưới đây là hàm chèn thêm một sinh viên mới vào đầu danh sách quản lý sinh viên ở ví dụ 4.10b:

```

void InserHead(List &L, dssv *p)
{
    if (L.pHead==NULL)
    {
        L.pHead = p;
        L.pTail = L.pHead;
    }
    else

```

```

{
    p->pNext = L.pHead;
    L.pHead = p;
}
}

```

e. **Chèn 1 nút mới vào cuối DSLKD** (độc giả tự làm, tương tự như mục d, song ở phía ngược lại)

#### f. Tìm một phần tử trong DSLKD

Trên DSLKD phải truy cập (access) tuần tự. Vì vậy ta chỉ dùng được thuật toán tìm kiếm tuần tự để xác định khóa cần tìm trong DSLKD.

```

void Tim(Bct_3 L, datatype *X)           //X: khóa cần tìm
{ Bct_2 *p;
    p = L.pHead;                      //cho p trả tới đầu DSLKD.
    while(p!=NULL && (p->Info.tp_i, X)!=0) //chừng nào
                                              chưa duyệt hết ds (ds khác rỗng)
        p=p->pNext;      // và dữ liệu của thành phần tp_i thuộc trường
        if (p==NULL) // Info không trùng với khóa X thì
            printf("Khong tim thay X trong ds."); // tìm tiếp
                                              (p=p->pNext); nếu p trả tới "hư vô"
        else          //thì tức là không tìm thấy khóa X,
            printf("Da thay X trong ds.",p->Info.tp_i);
                                              //còn ngược lại tìm thấy X
    }                                //i=1,2,...,k
}

```

*Ví dụ 4.14.* Trở lại DSLKD ở ví dụ 4.10b ta cài đặt hàm tìm một sinh viên theo khóa là tên của người ấy như sau:

```
void Tim(List L, char *X)
```

```
{
    dssv *p;
    p = L.pHead;
```

Ở ví dụ này, khóa X và dữ liệu của thành phần “ten” thuộc trường Info có kiểu char nên phải dùng hàm strcmp() để so sánh giá trị của chúng!

```
while(p!=NULL&&strcmp(p->Info.ten, X)!=0)
```

```

    p=p->pNext;
    if (p==NULL)
        printf("Khong tim thay Sv ten X trong lop.");
    else
        printf("Da thay Sv ten %s trong lop.",p-
              >Info.ten);
}

```

**g. Xóa (hủy) một phần tử có khóa X trong DSLKD.** Thuật toán thực thi thao tác này được mô tả bởi hàm sau:

```

void Del(Bct_3 &L, datatype *X)      //X: dữ liệu cần xóa
{ Bct_2 *p, *q;
  p=L.pHead; q=NULL;
  while (p!=NULL &&
         (p->Info.tp_i,X)!=0
  { q=p;
    p=p->pNext;
  }
  if(p==NULL) printf("Khong ton tai khoa X trong Ds.");
  else {
    if (q==NULL)
      { L.pHead=p->pNext;
        if(L.pHead==NULL)
          L.pTail=NULL; }
    else
      { q->pNext=p->pNext;
        if(p==L.pTail)
          L.pTail=q;
      }
    delete p;
  }
}

```

/\* Trước khi xóa,  
phải tìm xem có  
khóa cần xóa trong  
danh sách không. Nếu  
không có thì chẳng  
phải xóa gì cả! \*/

/\* Còn nếu có X nhưng con  
trỏ trung gian q trỏ tới chỗ  
trống thì tạo liên kết giữa  
phần tử kế tiếp với con trỏ  
pHead. Nếu danh sách rỗng  
thì cập nhật địa chỉ cho  
pTail \*/

/\* Ngược lại (q!=NULL) thì  
ghi nhớ địa chỉ của phần tử kế  
tiếp vào trường pNext của q.  
Nếu đã rà hết danh sách thì cho  
con trỏ pTail trỏ tới q rồi xóa X  
(bằng cách hủy vùng nhớ đã  
cấp cho p mà p lưu địa chỉ của  
vùng chứa X) \*/

*Ví dụ 4.15.* Trở lại DSLKD ở ví dụ 3.10b ta cài đặt hàm xóa tên (*kiểu char*) một sinh viên trong danh sách như sau:

```

void DelSv(List &L, char *X)
{
    dssv *p, *q;
    p=L.pHead; q=NULL;
    while (p!=NULL && strcmp(p->Info.masv, X) !=0)
    {
        q=p;
        p=p->pNext;
    }
    if (p==NULL) printf("Khong co sv voi ma x trong
                         Ds.");
    else
    {
        if (q==NULL)
            { L.pHead=p->pNext;
              if (L.pHead==NULL) L.pTail=NULL;
            }
        else
            { q->pNext=p->pNext;
              if (p==L.pTail)
                  L.pTail=q;
            }
        delete p;
    }
}

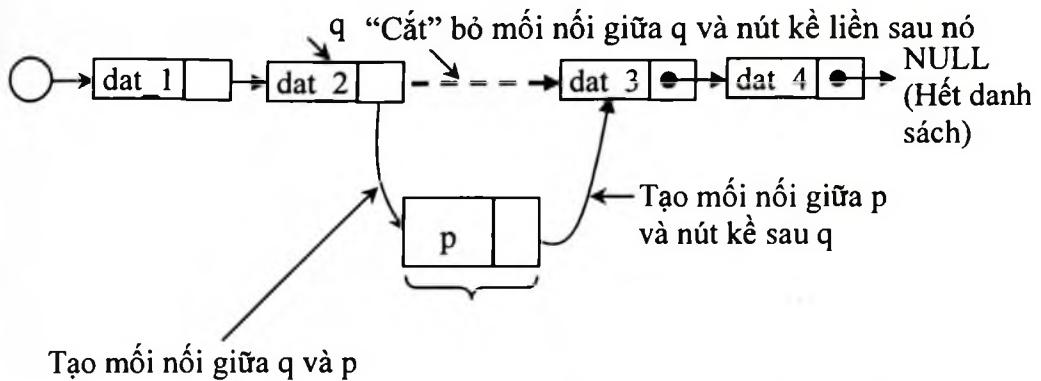
```

Việc dùng hàm này ở đây được lý giải như ở ví dụ 4.14 ngay trên!

#### *h. Chèn nút mới vào sau nút q trong DSLKD*

Công việc đặt ra là ta cần chèn nút mới vào sau nút q cho trước, cũng tức là chèn nó vào giữa 2 nút thuộc DSLKD.

Trước khi cài đặt thao tác trên bạn cần xem kỹ sơ đồ mô tả cơ chế làm việc của thao tác này như sau:



Hình 4.3. Sơ đồ mô tả thao tác chèn nút mới vào sau nút q trong DSLKD

Hàm mô tả thuật toán chèn nút mới vào sau nút q:

```

void Inser_After(List &L, Bct_2 *q, Datatype X)
{ Bct_2 *p=TaoNutMoi(X);           //p: là nút mới cần chèn
  if (p==NULL) return NULL;         //Nếu p rỗng thì hàm trả về
                                     giá trị rỗng (không chèn)
  if (q!=NULL)                     //Nếu q khác rỗng
  {
    p->pNext=q->pNext;           // thi tạo liên kết giữa p và q
    q->pNext=p;
    if (q==L.pTail)               //Nếu q trỏ tới cuối danh
                                   sách thì lưu địa chỉ của p vào con trỏ pTail
      L.pTail=p;
  }
  else                            //Ngược lại (q rỗng) thi
                                   chèn p vào đầu ds
    InserHead(List L, p);
}

```

#### i. Tìm trị max hoặc min của dữ liệu trong DSLKD

Đối với sinh viên, thao tác này “khó” hơn thao tác tìm một khóa bất kỳ trong danh sách. Dưới đây là hàm mô tả thuật toán này.

```

void DtbMax(List &L)
{ Bct_2 *p,*q,*max;

```

```

p=L.pHead;
while (p!=L.pTail)
{ max=p;
  q=p->pNext;
  while (q!=NULL)
  { if (q->Info.tp_k>max->Info.tp_k)
    max=q;
    q=q->pNext;
  }
  printf("In kết quả.");
  printf("\n");
  p=p->pNext;
}
}

```

*Ví dụ 4.16.* Trở lại DSLKD ở ví dụ 4.10b ta cài đặt hàm làm nhiệm vụ này như sau:

```

void DtbMax(List &L)
{ dssv *p,*q,*max;
  p=L.pHead;
  printf("Ten DTB:\n");
  while (p!=L.pTail)      //Chừng nào danh sách chưa rỗng thì
                           //làm các việc sau:
  { max=p;                //Giả định địa chỉ p lưu là địa chỉ chứa trị max
    q=p->pNext; //Lưu địa chỉ mà trả pNext trả tới vào trả trung gian q
    while (q!=NULL)        //Nếu q không rỗng thì
    { if (q->Info.DTB>max->Info.DTB) //Nếu trị của thành
                                             //phần DTB thuộc trường Info do con trả q
      max=q;                  //trả tới lớn hơn trị của tp DTB ở trường Info
                               //do con trả max trả tới thì
      q=q->pNext;           //max=q (lưu địa chỉ của q vào max) rồi xét
                               //phần tử kế tiếp để tìm được
    }
  }
}

```

```

) // địa chỉ đích thực chứa max đích thực (cho đến khi hết danh sách thì dừng)
printf("%s%.2f", max->Info.ten, max->Info.DTB); // In kết quả
    printf("\n");
    p=p->pNext;
}
}

```

### 4.3.3. Các phương án Sắp xếp DSLKD

Có hai phương án để làm việc này:

+ *Phương án thứ 1: Hoán vị dữ liệu ở trường Info.* Muốn thế ta có thể dùng các thuật toán sắp xếp đã học. Tuy nhiên khác với danh sách tĩnh (cài đặt bởi mảng) ở đây việc truy xuất vào các nút của DSLKD phải thông qua con trỏ liên kết chứ không phải thông qua index như mảng. Phương pháp này chỉ phù hợp với các trường Info có kích thước nhỏ, do nó cần dùng vùng nhớ trung gian. Với DSLKD có n phần tử thì ĐPT của phương pháp này là  $O(n^2)$ . Khi kích thước trường Info lớn, phương pháp này chiếm dụng nhiều vùng nhớ, tốc độ thực thi chậm.

+ *Phương án 2: Thay đổi các trường kiên kết.* Thực chất là thay đổi trình tự liên kết, không liên quan gì tới dữ liệu và do vậy cũng không phụ thuộc vào kích cỡ của trường info. Tuy nhiên việc thao tác trên trường liên kết gây nên phức tạp hơn phương pháp 1 khi cài đặt (cần kỹ năng lập trình cao hơn).

Một trong các cách thay đổi trình tự liên kết là tạo một danh sách mới (có trật tự khác) từ danh sách ban đầu. Ký hiệu danh sách mới này là NewL. Ta dùng một trong các thuật toán sắp xếp đã học, chẳng hạn ta sử dụng thuật toán sắp xếp lựu chọn cho phương án 2. Dưới đây là mô tả quy trình của phương án 2:

Bước 1: Khởi trị NewL rỗng;

Bước 2: Tìm phần tử min trong danh sách cũ;

Bước 3: Tách min ra khỏi danh sách cũ;

Bước 4: Chèn min vào cuối ds NewL;

Lặp lại bước 2 khi chưa xét hết ds cũ.

Cài đặt phương án 2:

```

void Varian2_SeleSort(List &L)
{ List dsMoi;
  Bct_2 *min, *p, *q, *min_befo;
  dsMoi.Phead=dsMoi.pTail=NULL; //Khởi trị cho danh sách
                                 mới là NULL
  while (L.pHead!=NULL //Chừng nào danh sách khác rỗng thì lặp:
  { p=L.pHead;//Lưu địa chỉ pHead vào p (cho p trỏ tới đầu danh sách)
    q=p->pNext;      //Lưu địa chỉ của nút kế tiếp vào q
    min=p;            //Lưu địa chỉ của con trỏ p min
    min_befo=NULL;    //Khởi trị cho con trỏ min_befo
                       (min_befo: con trỏ trỏ vào địa chỉ
    while (q!=NULL)   //có trị min ban đầu)
    { if(q->info<min->Info) //Nếu địa chỉ trường Info
       do q trỏ tới thấp hơn địa chỉ trường Info
       min=q; min_befo=p; //do min trỏ tới thì địa chỉ
                           của min lúc này là địa chỉ của q
       p=q;                //lưu địa chỉ của q vào p
       q=q->qNext;        //xét nút kế tiếp
       if(min_befo!=NULL)  //Nếu địa chỉ của min_befo
                           khác rỗng thì
         min_befo->pNext=min->pNext; //lưu địa chỉ của
                                         trường pNext của min vào
         else                  //trường pNext của min_befo
           L.pHead=min->pNext; //Ngược lại lưu địa chỉ của
                               pNext của con trỏ min vào pHead
         min->pNext=NULL;//Cho trỏ min trỏ tới cuối danh sách
         Inse_Tail(dsMoi, min); //rồi chèn địa chỉ của
                               min vào cuối danh sách mới
    }
  L=dsMoi;
}

```

#### 4.3.4. Một số thuật toán sắp xếp dữ liệu của DSLKD

Với DSLKD có hai thuật toán sắp xếp hiệu quả nhất là Quick Sort và Merge Sort. Ta lần lượt phân tích cơ chế làm việc và cài đặt hai thuật toán đó trên DSLKD.

Khi nghiên cứu mảng (một kiểu cấu trúc tĩnh) độc giả đã thấy rõ cài đặt Quick Sort và Merge Sort cho mảng khá phức tạp về kỹ năng lập trình. Tuy nhiên với DSLKD (một kiểu dữ liệu động) thì cài đặt hai thuật toán này đơn giản hơn nhiều và cả hai không đòi hỏi vùng nhớ trung gian.

##### a. Sắp xếp DSLKD nhờ Quick Sort

Cho trước một DSLKD lưu các số nguyên, hãy xếp ds này tăng dần nhờ Quick Sort:

7 -10 21 1 60 15 -3

Chúng ta ôn lại ý tưởng chính của Quick Sort:

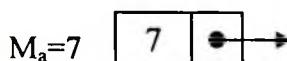
Chọn một phần tử (*tùy ý*) làm mốc. Theo thuật toán ở mục 3.2.7, chia dãy đã cho thành ba dãy:

- Dãy chứa các trị nhỏ hơn M;
- Dãy chứa các trị =M
- Dãy chứa các trị lớn hơn M

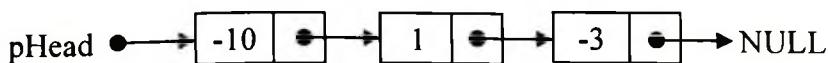
Lặp lại 3 thao tác trên cho tới khi chỉ còn một phần tử duy nhất của dãy thì dừng.

Với Linked List đã sẵn có con trỏ pHead, nên tiện lợi nhất là chọn phần tử đầu danh sách làm mốc. Việc chọn bất kỳ phần tử nào khác cũng làm phức tạp khi thảo chương và tăng thời gian thực hiện chương trình khi độ dài danh sách đạt hàng ngàn, hàng vạn... Dưới đây là các pha làm việc của thuật toán này:

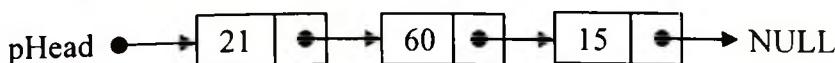
Pha 1: - Chọn mốc  $M_a=7$ ;

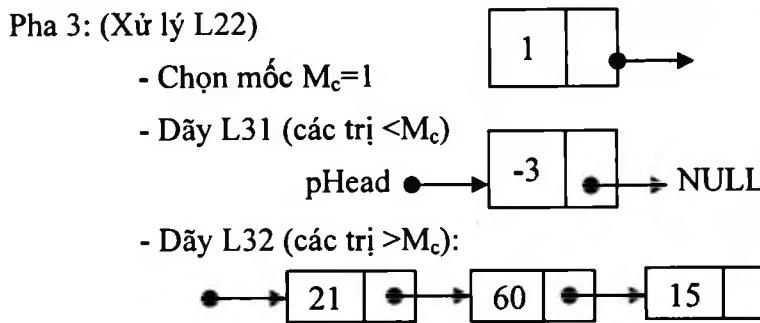
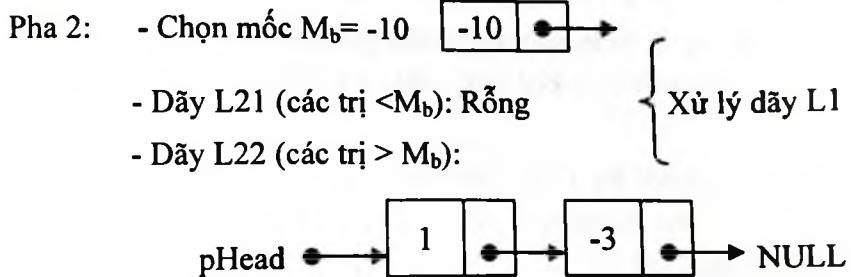


- Dãy con L1 (các trị < M)

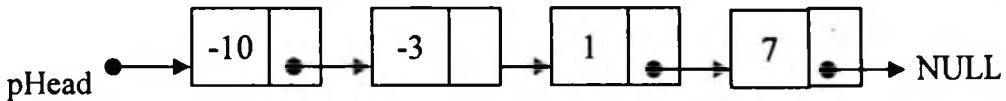


- Dãy con L2 (các trị > M)

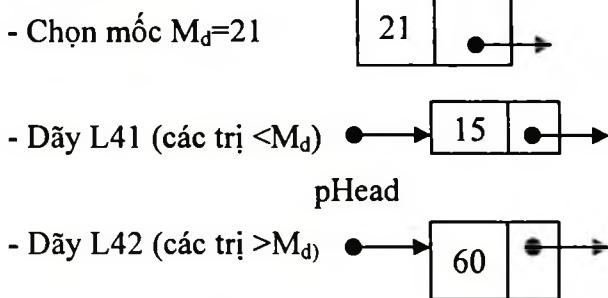




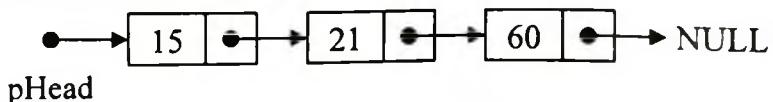
- *Sắp xếp 1:* Ghép nối L31 với  $M_b = -10$  (về phía phải-cuối-của  $M_b$ ) rồi với  $M_c = -3$  rồi với  $M_a = 7$  (cũng về phía cuối) ta được dãy Kq1:



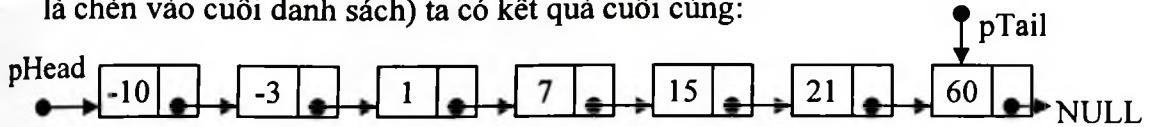
Pha 4: (Xử lý L32):



- *Sắp xếp 2:* Ghép nối  $M_d$  với L41 (về phía cuối) rồi với L42 (cũng về phía cuối của dãy ( $M_d$ , L41)) được dãy Kq2:



- *Sắp xếp 3:* Ghép nối dãy Kq2 với dãy Kq1 (về phía cuối của Kq1, tức là chèn vào cuối danh sách) ta có kết quả cuối cùng:



Hàm mô tả Quick Sort Algorithm để sắp xếp DSLKD:

```

void Lkd_Qsort( List &L)
{
    //dc: địa chỉ, ds: danh sách
    Bct_2 *p, *M ; //M: mốc (để phân hoạch (chia)) dãy hiện hành
    List L1,L2;
    if (L.pHead==L.pTail) return;//Ds đã có trật tự, trả quyền
        //điều khiển cho main()
    L1.pHead==L1.pTail=NULL;      //Khởi trị ds L1
    L2.pHead==L2.pTail=NULL;      //Khởi trị ds L2
    M=L.pHead;                  //Chọn mốc là nút đầu dãy
    L.pHead=M->pNext//Lưu địa chỉ của trường pNext do M trả tới vào pHead
    while (L.pHead!=NULL) //Chừng nào chưa xét hết danh sách, lặp:
    {
        p=L.Head;            //Cho p trả tới đầu danh sách L
        L.pHead=p->pNext;   //Lưu dc trường pNext do p trả tới vào con
        //trở đầu danh sách
        p->pNext=NULL;      //Hết danh sách L
        if (p.Info<=M.Info)  //Nếu data ở trường Info của p không
            //lớn hơn data ở trường Info
            Inser_Tail(L1,p); //của M thì gọi Inser_Tail để nối nút p
            //vào cuối danh sách con L1
        else     InserTail(L2,p); //Ngược lại gọi Inser_Tail để nối nút p
            //vào cuối L2.
    }
    List_Qsort(L1); //Gọi đệ quy List_Qsort() để xếp danh sách con L1
    List_Qsort(L2); //Gọi đệ quy List_Qsort() để xếp danh sách con L2
}

```

//Đoạn chương trình ghép nối L1 với mốc M và với L2 để có danh sách lớn đã xếp:

```

if (L1.pHead!=NULL) ;
{
    L.pHead=L1.pHead;
    L1.pTail->p.pNext=M;
}
else
{
    L.pHead=M;
    M.pNext=L2;
    if (L2.pHead!=NULL)
        L.pTail=L2.pTail
    else
        L.pTail=M;
}

```

### b. Sắp xếp DSLKD nhờ Merge Sort

Trước hết chúng ta ôn lại một vài kiến thức liên quan tới Merge Sort và nội dung chính của thuật toán này.

+ Run là dãy con có trật tự thuộc dãy đã cho. Dãy có độc nhất một phần tử cũng là Run.

+ Quy trình làm việc của Merge Sort tựu chung gồm 4 bước sau:

Bước 1: Phân phối luân phiên các Run lên các danh sách con L1, L2;

Bước 2: Nếu chưa xét hết L1 thì gọi đệ quy MergeSort(L1) để xếp L1;

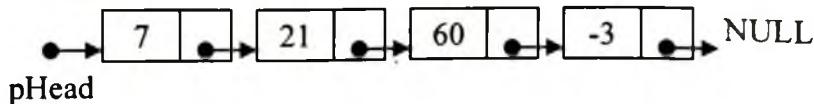
Bước 3: Nếu chưa xét hết L2 thì gọi đệ quy MergeSort(L2) để xếp L2;

Bước 4: Trộn (Merge) hai ds L1 và L2 để được DSLKD L có trật tự.

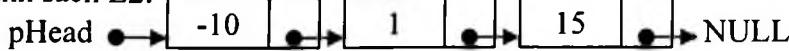
Cho DSLKD chứa các số nguyên 7 -10 21 1 60 15 -3 → Để tiện ký hiệu danh sách này là A

+ *Lần phân phối 1:*

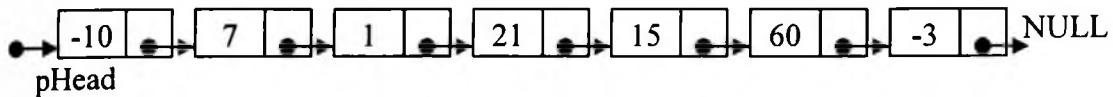
- Danh sách L1:



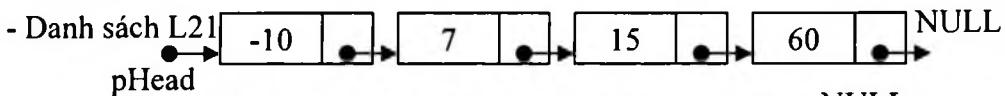
- Danh sách L2:



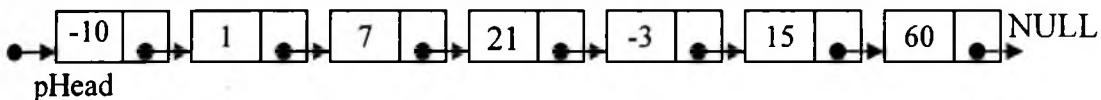
+ Trộn lần 1: Trộn L1 và L2 được ds mới (vẫn lưu vào A):



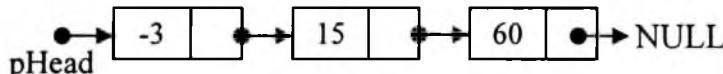
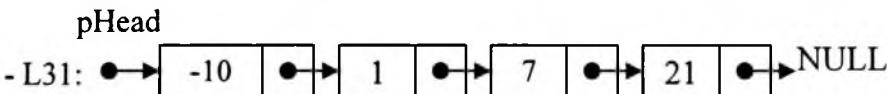
+ Lần phân phối 2:



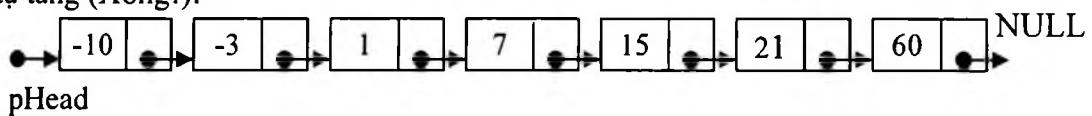
+ Trộn lần 2: Trộn L21 và L22 được ds mới (vẫn lưu vào A):



+ Lần phân phối 3:



+ Trộn lần 3: Trộn L31 và L32 được kết quả là một DSLKD xếp theo trật tự tăng (Xong!):



Hàm mô tả Merge Sort để sắp xếp DSLKD

+ Modul 1:

```

void XepDs(List &L)
{
    List L1, L2;
    if (L.pHead==L.pTail) return; //Danh sách đã có trật tự
    L1.pHead==L1.pTail=NULL; { //Khởi trị cho các danh sách con
    L2.pHead==L2.pTail=NULL; }
    PhanphoiRun(L, L1, L2); //Phân phối các Run từ ds đã cho
    xuống các danh sách con L1, L2
}
  
```

```

        XepDs (L1);           //Gọi đệ quy XepDs() xếp danh sách L1
        XepDs (L2);           //Gọi đệ quy XepDs() xếp danh sách L2
        Tron (L, L1, L2);    //Trộn L1 và L2
    }

+ Modul 2:

void PhanphoiRun(List &L, List &L1, List &L2)
//Hàm thực thi phân phối các Run
{
    Bct_2 *p;
    do
        { p=L.Head;           //Đoạn chương trình phân hoạch L
          L.pHead=p->pNext;   //Cho p trỏ tới đầu danh sách L
          p->pNext=NULL;      //Duyệt các phần tử của danh sách đến
          InserTail (L1,p);   //Chèn (thêm) p vào cuối danh sách
        }
    while ((L.pHead) && (p->Info<=L.pHead->Info));
//Lặp lại các việc trên cho đến khi điều kiện
    if (L.pHead)           //viết sau while được thỏa mãn thì dừng
        PhanphoiRun (L, L1, L2); //Nếu con trỏ pHead trỏ
                           //tới đầu danh sách thì bắt đầu phân hoạch L
    else p.Tail=NULL; //thành 2 danh sách con L1, L2 (tức là
                      //phân phối các Run xuống L1, L2)
}
//còn không thì kết thúc

+ Modul 3:

void Tron(List &L, List &L1, List &L2)
{
    Bct_2 *p;
    while ((L1.pHead) && (L2.pHead))
    { if (L1.pHead->Info <= L2.pHead->Info)
        { p=L1.pHead;
          L1.pHead=p->pNext;
        }
    }
}

```

```

    }
    else
    { p=L2.pHead;
      L2.pHead=p->pNext;
    }
    p->pNext=NULL;
    InserTail(L,p);
}
if (L1.pHead)
{ L.pTail->pNext=L1.pHead;
  L.Ptail=L1.pTail
}
else { if (L2.pHead)
        L.pTail->pNext=L2.pHead;
        L.pTail=L2.pTail;
      }
}

```

### c. Sắp xếp DSLKD nhờ Radix Sort

Trước khi đọc tiếp mục này, độc giả cần xem lại mục 2.14. Tại mục này, thực chất hoạt động của Radix sort đã được trình bày một cách trực quan và dễ hiểu.

#### Các bước của thuật toán Radix Sort

Bước 1: Khởi trị cs=0; // cs: chữ số để phân loại; cs=0 → chữ số hàng đơn vị; cs=1 → chữ số hàng chục; cs=2 → chữ số hàng trăm;...

Bước 2: Tạo các lô B<sub>0</sub>, B<sub>1</sub>, . . . , B<sub>9</sub> và khởi trị tất cả chúng là rỗng.

Bước 3: Duyệt từ đầu dãy phần tử đã cho đến hết dãy làm: Đặt số a<sub>i</sub> vào lô B<sub>k</sub> với k là chữ số thứ k của a<sub>i</sub>.

Bước 4: Ghép các số ở các lô B<sub>0</sub>, B<sub>1</sub>, . . . , B<sub>9</sub> với nhau (theo trình tự thỏa mãn trật tự xếp của ta) để thu được dãy A có trật tự.

Bước 5: + Tăng cs=cs+1;

+ Nếu  $cs < m$  ( $m$ : các chữ số có thể có thuộc các số trong dãy đã cho:  $0 \leq m \leq 9$ ) thì quay lại Bước 2;

+ Còn thì dừng;

Dưới đây là 3 modul chủ chốt để sắp xếp theo Radix Sort (các modul vẫn dùng DSLKD do vậy độc giả cần xem lại phần khai báo cấu trúc DSLKD tại mục 4.3.2.a).

+ *Hàm thực thi Radix Sort Algorithm*

```
void RadixSort(List &L, int m)
{
    List b[10];
    Bct_2 *p; int i, k; //k là chỉ số của các lô b[]
    if (L.pHead==L.ptail) return ;
    for(i=0; i<10; i++)
        b[i].pHead=b[i].pTail=NULL;
    for(k=0; k<m; k++)
        {while (L.pHead)
            {L.pHead=p->pNext; pNext=NULL;
             i=Getdigit(p->pInfo, k);
             AddTail(b[i], p);
            }
        L=b[0];
        for (i=1, i<10; i++)
            InserTail(L, b[i]); //Nối b[i] vào cuối danh sách L
        }
}
```

+ *Hàm chèn data vào cuối danh sách*

```
Bct_2* InsertTail(L &L, int x) //Hàm chèn data vào cuối
danh sách
{ Bct_2* pt_moi=GetNode(x);
  if(pt_moi==NULL) return NULL;
  if(L.pHead==NULL)
```

```

    {
        L.pHead=pt_moi; L.pTail=L.pHead;
    }
else
{
    L.pTail->Next=pt_moi;
    L.pTail=pt_moi;
}
return pt_moi;
}

```

+ *Hàm tạo các chữ số k để phân loại*

```

int GetDigit(unsign long n, int k)
{
    switch(k)
    {
        case 0: return (n%10); break;
        case 1: return ((n%10)%10); break;
        case 2: return ((n/100)%10); break;
        case 3: return ((n/1000)%10); break;
        case 4: return ((n/10000)%10); break;
        case 5: return ((n/100000)%10); break;
        case 6: return ((n/1000000)%10); break;
        case 7: return ((n/10000000)%10); break;
        case 8: return ((n/100000000)%10); break;
        case 9: return ((n/1000000000)%10); break;
    }
}

```

#### 4.3.5. Cài đặt hoàn chỉnh DSLKD cho một áp dụng thực tế

Ở mục 4.3.2 chúng ta đã phân tích chi tiết và thiết kế từng modul ứng với mỗi thao tác trên DSLKD. Tuy nhiên với nhiều sinh viên việc lắp ghép các modul có sẵn vào với nhau để có một chương trình hoàn chỉnh là khá chật vật. Thậm chí ngay từ khởi đầu của việc cài đặt là định nghĩa (khai báo) DSLKD đã

có khá nhiều bạn sai sót. Sau đây là cài đặt một DSLKD thực hiện một số các thao tác trên các nút của nó, gồm:

- Khởi trị cho ds (1)
- Tạo nút (node) mới cho danh sách (2)
- Chèn một nút mới vào danh sách (3)
- Tìm một nút với khóa cho trước tùy ý X (4)
- Tìm giá trị lớn nhất của dữ liệu (5)
- Tìm một giải giá trị dữ liệu thỏa mãn điều kiện cho trước (6)
- Duyệt danh sách (chức năng này phải dùng được cho cả các chức năng (3), (4),(5),(6),(8) (7))
- Xóa một nút trong danh sách (8)

### **Chương trình**

```
#include "conio.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

typedef struct
{
    char ten[25];
    char masv[7];
    float DTB;
} SV;

typedef struct Qlsv
{
    SV Info;
    struct Qlsv *pNext;
} dssv;

typedef struct ds
{
    dssv *pHead;
    dssv *pTail;
} List;
```

```
void Init(List &L)      //Hàm khởi trị cho danh sách
{
    L.pHead=NULL;          //Ban đầu danh sách rỗng
    L.pTail=NULL;
}

dssv *TaoNutMoi(SV x)
{ dssv *NutTg;
    NutTg=new dssv; //Hàm cấp phát vùng nhớ cho biến động NutTg
    if(NutTg==NULL)
    {
        printf("khong du bo nho");
        exit(1);
    }
    NutTg->Info =x //Lưu dữ liệu x vào trường Info của Nút trung gian
    NutTg->pNext = NULL; //Chưa có Nút kế tiếp
    return NutTg;
}

void AddHead(List &L, dssv* p) //Hàm chèn 1 Nút mới vào
                                //đầu danh sách
{
    if (L.pHead==NULL)
    {
        L.pHead = p;
        L.pTail = L.pHead;
    }
    else
    {
        p->pNext = L.pHead;
        L.pHead = p;
    }
}
```

```
    }
}

//Tim 1 sinh vien co ten x trong lop hoc
void Tim(List L, char *X)
{ dssv *p;
  p = L.pHead;
  while(p!=NULL&&strcmp(p->Info.ten, X)!=0)
    p=p->pNext;
  if (p==NULL)
    printf("Khong tim thay Sv ten X trong lop.");
  else
    printf("Da thay Sv ten %s trong lop.",p-
          >Info.ten);
}

void DtbMax(List &L)      //Hàm tìm giá trị lớn nhất của dữ liệu
{ dssv *p,*q,*max;
  p=L.pHead;
  printf("Ten DTB:\n");
  while (p!=L.pTail)
  { max=p;
    q=p->pNext;
    while (q!=NULL)
    { if (q->Info.DTB>max->Info.DTB)
        max=q;
      q=q->pNext;
    }
    p=p->pNext;
  }
  if (q->Info.DTB=max->Info.DTB)
```

```
        printf("%s %.2f", max->Info.ten, max->Info.DTB);
        printf("\n");
    }

void Duyet(List L)      //In toàn bộ danh sách lớp lên màn hình
{ dssv *p;
    printf("Ho ten   Dtb \n\n");
    p=L.pHead;
    while(p!=NULL)
    {
        //puts(p->Info.ten); Có thể dùng lệnh này sẽ gọn hơn
        printf("%s\t%.2f\n", p->Info.ten, p->Info.DTB);
        p=p->pNext;
    }
}

void Dsdat(List L)    //Tìm và In danh sách các SV có DTB>=5
{ dssv *p;
    p=L.pHead;
    printf("Hoten   DTB:\n");
    while (p!=NULL)
    {if (p->Info.DTB>=5)
        {//puts(p->Info.ten);          //Có thể dùng lệnh này
         thay vì dùng lệnh printf
        printf("%s\t%.2f", p->Info.ten, p->Info.DTB);
        printf("\n");
    }
    p=p->pNext;
}
}
```

```

void Swap(SV &s1, SV &s2)           //Hàm đổi chỗ, dùng
{ SV tam;                         //trong sắp xếp khi
    tam=s1;                         //phát hiện nghịch thế
    s1=s2;
    s2=tam;
}

void SeleSort(List &L)             //Xếp danh sách bằng
{ dssv *p, *q, *min;               //phương pháp lựa chọn
    p=L.pHead;
    while(p!=L.pTail)
    {
        min=p;
        q=p->pNext;
        while(q!=NULL)
        {
            if(q->Info.DTB<p->Info.DTB)
                min=q;
            q=q->pNext;
        }
        Swap(min->Info, p->Info);
        p=p->pNext;
    }
}

void DelSv(List &L, char *X)      //Hàm xóa tên một sinh viên
{ dssv *p, *q;                   theo khóa X tùy ý cho trước
    p=L.pHead; q=NULL;
    while (p!=NULL && strcmp(p->Info.masv, X) !=0)

```

```

{ q=p;
  p=p->pNext;
}
if(p==NULL) printf("Khong co sv voi ma x trong
                     Danh sach.");
else
{
  if (q==NULL)
  { L.pHead=p->pNext;
    if(L.pHead==NULL) L.pTail=NULL;
  }
  else
  { q->pNext=p->pNext;
    if(p==L.pTail)
      L.pTail=q;
  }
  delete p;
}
}

void main() //Hàm chính của chương trình
{
  List L1;
  dssv *p;
  char ten[25];
  float dtb,Max=1e-6; int d=0;
  char ms[7];
  SV s; clrscr();
  Init(L1);
  do
  { ten[0]='\0';
    printf("Nhập tên (Ngưng nhập go ENTER):");
    fflush(stdin); //Xóa sạch vùng đệm bàn phím
    //trước khi nhập tên
  }
}

```

```
gets(s.ten);
if(strlen(s.ten)>0)
{
    printf("Nhập mã số:");
    gets(s.masv);
    printf("Nhập điểm trung bình:");
    scanf("%f",&dtb);
    printf("= = = = = = = = = =\n");
    s.DTB=dtb;
    p=TaoNutMoi(s);
    AddHead(L1,p);
}
}while(strlen(s.ten)>0);
printf("Danh sách sinh viên vừa nhập:\n");
Duyet(L1);
printf("= = = = = = = = = =\n");
printf("Nhập tên cần tìm:");
gets(ten);
printf("Kết quả tìm:\n");
Tim(L1,ten);
printf("\n");
printf(" Go phím bất kỳ để tiếp tục...\n");
getch();
SeleSort(L1);
printf("Danh sách đã xếp theo Dtb tang:\n");
Duyet(L1);
printf("= = = = = = = = = =\n");
printf("Danh sách các SV có DTB>=5:\n");
Dsdat(L1);
printf("= = = = = = = = = =\n");
printf("Ds các sv có DTB cao nhất là:\n");
```

```

DtbMax(L1);
printf("\n");
printf("Nhập mã SV cần xoá:"); gets(ms);
DelSv(L1, ms);
printf("Ds sau khi xoá một SV:\n");
Duyet(L1);
printf("Done!");
getch();
}

```

*Mở rộng:* Cũng đề bài như trên, bạn đọc hãy tổ chức các thao tác trên thành Menu để người dùng tự chọn.

## 4.4. CÁC DANH SÁCH LIÊN KẾT ĐƠN ĐẶC BIỆT

### 4.4.1. Kiểu dữ liệu trừu tượng Ngăn xếp

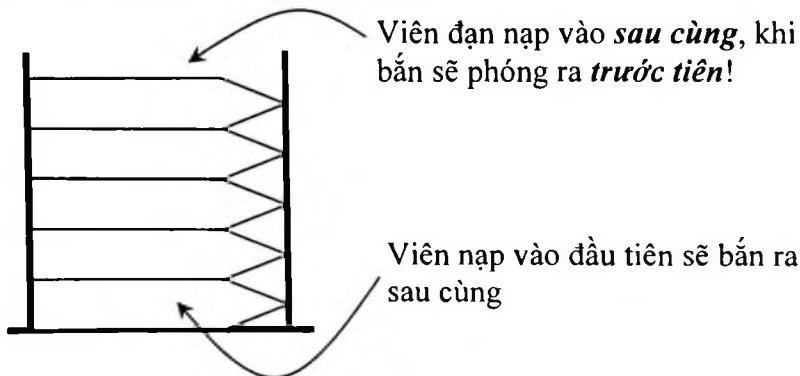
**Kiểu dữ liệu trừu tượng (Stack ADT)** - Tên khác: ds “Vào Sau Ra Trước” (Last In First Out: LIFO). Để gọn từ nay chúng ta sẽ dùng tên STACK.

a. **Định nghĩa:** STACK là một trường hợp đặc biệt của danh sách tuyến tính, song STACK làm việc theo cơ chế:

+ Vào sau ra trước (Last In First Out)

+ Bất kỳ lúc nào cũng có thể nạp thêm phần tử mới vào nó, tuy nhiên khi lấy ra thì: Chỉ có thể **lấy ra** phần tử **nạp vào sau cùng** (để gọn về sau thay vì nói “nạp vào” ta nói “vào”)

b. **Hình ảnh thực tế của STACK:** Để giúp độc giả dễ hiểu danh sách STACK, chúng tôi đưa ra một hình ảnh thực tế của nó: Đó là những hộp đạn của các loại súng bộ binh, chẳng hạn hộp đạn súng lục:

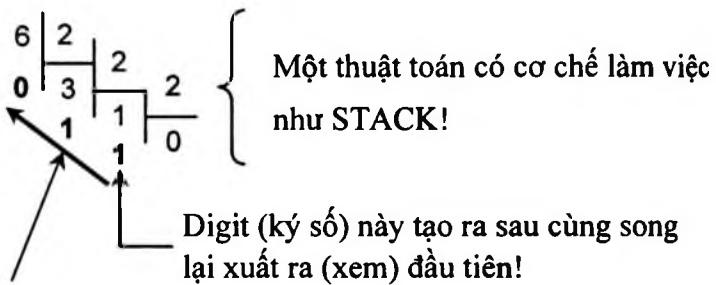


Hình 4.4. Một hình ảnh thực tế của STACK

Rõ ràng STACK chỉ có một lối duy nhất để *dữ liệu vừa vào vừa ra* (Nạp và lấy dữ liệu ra tại cùng một đầu), lối này người ta quen gọi là Đỉnh (Top) của STACK.

### c. Một vài ứng dụng của STACK cho toán học

Đó là thuật toán đổi một số nguyên từ hệ thập phân sang hệ nhị phân. Như chúng ta đã học, để làm việc này ta chia số nguyên ở hệ thập phân liên tiếp cho cơ số 2 của hệ nhị phân, việc này dừng khi thương số bằng không. Nguyên tắc ghi kết quả phép chuyển đổi trên đây: đọc các số dư của phép chia ấy ngược từ dưới lên (xem ví dụ bên dưới). Điều đó nghĩa là: số dư tính được cuối cùng sẽ được ghi (xuất) *đầu tiên*. Đó chính là cơ chế làm việc của STACK.



Chiều đọc kết quả ( $6_{10}$  đổi sang hệ nhị phân là  $110$ )

*c<sub>1</sub>*. Vì vậy một ứng dụng của LIFO là dùng nó để chuyển đổi một số ở hệ thập phân sang hệ nhị phân. (Xem code thực thi thuật toán trên tại hàm void deci\_bina(int n) ở mục 4.4.1/d.2.8 cài đặt Stack bằng DSLKD).

*c<sub>2</sub>*. Dùng Stack chuyển đổi một biểu thức Infix (trung tố) thành biểu thức Postfix (hậu tố).

Ta gọi các biểu thức viết theo lối thông thường là biểu thức Infix (biểu thức Trung tố, các biểu thức có ngoặc đơn). Ví dụ:  $3*(4+5)$  là biểu thức Infix. Việc tính toán nhờ máy tính khi dùng các Infix không thuận tiện bằng các biểu thức Posfix (đặc biệt khi biểu thức đã cho chứa quá nhiều ngoặc đơn).

Còn một biểu thức được gọi là Postfix (biểu thức Hậu tố) nếu các toán hạng (Operand) trong nó đều viết ở bên trái tất cả các toán tử (Operator) và trong Postfix *không có các ngoặc đơn*. Thứ tự các toán tử trong Postfix viết theo *thứ tự ngược lại* so với vị trí của chúng trong Infix (Còn các toán hạng giữ nguyên trình tự như trong Infix đã cho). Ví dụ: ứng với Infix nêu trên ta có biểu thức Postfix là  $3\ 4\ 5\ +\ *$ . Vì vậy Posfix còn có tên gọi khác là biểu thức RPN (Reverse Polish Notation, tạm dịch: ký pháp Ba lan đảo ngược).

Các biểu thức Prefix thì ngược lại với Postfix: các toán tử viết ở bên trái các toán hạng và cũng không chứa các ngoặc đơn. Ví dụ:  $* + 3 4 5$ .

Dưới đây là bản code ngắn gọn viết trên C++ thực thi thuật toán chuyển đổi Infix sang Postfix.

```

int main()
{
    stack<char> oper;
    string s;
    cout<<"Nhập một biểu thức Infix:"
    getline(cin, s);
    istringstream in(s);
    char c;
    while (in>>c)
    {
        if(c=='+' || c=='-' || c=='*' || c=='/')
            oper.push(c);
        else if (c=='(' || c== ')')
            { cout<<oper.top()<<" ";
            oper.pop();
            }
        else if(c>='0' || c<='9')
            { in.putback(c);
            int n;
            in>>n;
            cout<<n<<" ";
            }
    }
    cout<, oper.top()<<endl;
}

```

Hãy thử lần 1 với Infix:  $30*(40+50)$ , tất nhiên kết quả phải là Postfix:  $30\ 40\ 50\ +\ *$

*Phân tích đoạn code trên đây:*

Ở lần lặp thứ nhất, while trích xuất ký số 3 ở trên dòng (line) vào biến c. Vì nó thất bại trong 2 lần thử if đầu tiên, nhưng vượt qua lần thử thứ 3 ( $c \geq '0' \parallel c \leq '9'$ ), nó đặt 3 trở lại vào dòng để nó có thể đọc số 3 với ký tự thứ hai dưới dạng số nguyên 30 và sau đấy nó sẽ xuất. Ở lần lặp thứ 2, while trích xuất ký tự trên dòng vào c. Vì nó vượt qua các lần thử if đầu tiên ( $c == '*'$ ) nên nó đẩy \* lên trên stack oper để \* có thể được xuất sau đó. Ở lần lặp thứ 3, ký tự (được trích xuất từ dòng vào c. Vì nó thất bại trong 3 lần kiểm thử if, nó bị bỏ qua). Ở lần lặp thứ 4, số nguyên 40 được trích xuất bằng cùng một cơ chế đã trích xuất số 30 (ở lần lặp thứ nhất). Trước tiên máy đọc số 4 rồi đẩy nó lên phía trên stack và sau đó đọc số nguyên 40, rồi 40 được xuất. Ở lần lặp kế tiếp ký tự + được đọc và đẩy lên phía trên stack. Rồi đến 50 được đọc và xuất cũng theo cách trên. Cuối cùng ký tự ')' được đọc làm cho + được đẩy ra khỏi stack và in, rồi đến lượt \* được đẩy ra và in. Chú ý code mô tả trên đây đòi hỏi Infix nhập vào phải đặt bên trong các dấu ngoặc đơn. Hãy kiểm thử lần 2 code trên với Infix khác như sau:  $(80-30)*(40+(10*50))$ .

*Chú ý:*

Trước khi sử dụng code trên bạn phải biết các thành phần của template lớp đối tượng chứa stack của C++ chuẩn là:

```

typedef enum { true, false } bool;

template <class T> class stack
{ public:
    stack();                                //Hàm tạo mặc định
    stack(const stack&);                   //Sao chép hàm tạo
    ~stack();                                //Hàm hủy tạo
    stack& operator=(const stack&);          //Đăng ký toán tử
    int size() const;                        //Số phần tử của stack
    bool empty() const;                      //Trả về true nếu stack rỗng
    T& top();                               //Trả về phần tử ở đỉnh stack;
    void push(const T&);                  //Nạp phần tử từ đỉnh vào stack

```

```

void pop()      //Đẩy (lấy) một phần tử ra khỏi stack
                // v.v...
};


```

*c<sub>3</sub>. Một số ứng dụng khác của STACK:* Nó lưu các đỉnh của đồ thị khi tìm kiếm trên đồ thị theo thuật toán DFS; khử đệ quy; trong các trình biên dịch, LIFO được dùng để lưu môi trường làm việc của các hàm hoặc các thủ tục; v.v...

#### *d. Các công cụ cài đặt STACK*

Có hai cách cài đặt ds STACK: Dùng mảng 1\_chiều và dùng DSLKD:

*d<sub>1</sub>. Dùng mảng 1\_chiều:* Cách này đơn giản trong lập trình, nhưng tốn bộ nhớ nếu kích thước STACK lớn và thời gian thực thi chương trình sẽ lớn. Chúng tôi chuyển mục này thành bài tự luyện cho các bạn.

*d<sub>2</sub>. Dùng DSLKD:* Ta sẽ mô tả các thao tác trên tập các dữ liệu của ds STACK bằng C/C<sup>++</sup> như sau:

##### *d<sub>2.1</sub>. Định nghĩa STACK dùng DSLKD*

```

struct node
{
    int info;
    node* next;
};

```

##### *d<sub>2.2</sub>. Khởi tạo cho STACK*

```

void init(node* &ptop)
{
    ptop=NULL;           //Khởi đầu, danh sách STACK rỗng
}

```

##### *d<sub>2.3</sub>. Kiểm tra xem STACK có rỗng không*

```

int empty(node* &ptop)
{
    return(ptop==NULL);
}

```

##### *d<sub>2.4</sub>. Nạp (Push) phần tử mới (nút mới) vào STACK*

```
void napvao(node* &ptop, int x) //Nạp nút mới vào STACK
```

```

{ node *p;
  p = new node;           //Cấp phát vùng nhớ động cho con trỏ p
  p->info=x;p->next=ptop;    //nạp phần tử x vào trường
                                info từ định STACK
  ptop=p;                  //lưu địa chỉ của p vào con trỏ định
} ;

```

d<sub>2.5</sub>. Lấy (hủy, Pop) một phần tử ra khỏi ds STACK

```

int layra(node* &ptop) //Lấy một phần tử ra khỏi
                        (từ đỉnh) STACK
{ assert(!empty(ptop)); //Xác định STACK không rỗng (thì)
  int x=ptop->info;    //lấy dữ liệu từ trường info của con
                        trò đỉnh lưu vào biến x
  node *p = ptop; //Lưu địa chỉ con trò đỉnh vào biến động *p
  ptop=ptop->next; //Con trò đỉnh trả tới phần tử kế tiếp
  delete p;           //Giải phóng p
  return(x); //Hàm layra() trả về phần tử x - phần tử cần hủy
}

```

#### *d<sub>2.6</sub>. Xem một phần tử ở đỉnh STACK*

```

int ViewTop(node * &ptop)
{ assert(!empty(ptop)); //Nếu Stack không rỗng thì nạp
  int x=ptop->info;    //dữ liệu ở trường info vào x
  return(x);             //trả về dữ liệu ở đỉnh Stack
};                      //Hàm assert dùng để kiểm tra ds có rỗng không, hàm này
                        //nằm trong hàm tiêu đề assert.h sẵn có của C/C++

```

*d<sub>2.7</sub>. Xóa STACK (Giải phóng toàn bộ các nút của danh sách)*

```

void clear(node* &ptop)
{ node *p,*p1;
  p=ptop; //Cho p trỏ tới đỉnh STACK
  while(p!=NULL) //Chừng nào mà danh sách
    không rỗng thì lặp các việc sau:

```

```

    { p1=p;           //Cho p trỏ tới địa chỉ của p
      p=p->next;    //Xét nút kế tiếp
      delete p1;      //Giải phóng (hủy) p1
    }
    ptop=NULL;        //STACK đã bị xóa sạch
  }

```

*d<sub>2.8</sub>. Dùng DSLKD cài đặt (set up) STACK*

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <assert.h>
#define true 1
#define false 0
//-----
struct node
{
  int info;
  node* next;
};
//-----
//Khoi tao NganXep, dat ptop=NULL.
void init(node* &ptop)
{
  ptop=NULL;
}
//-----
//Xoa Ngan xep, tuc la giai phong tat ca cac nut trong NganXep.
void clear(node* &ptop)
{
  node *p, *p1;
  p=ptop;

```

```
while(p!=NULL)
{ p1=p;
  p=p->next;
  delete p1;
}
ptop=NULL;
}

//-----
//Kiem tra xem NganXep co rong khong.
int empty(node* &ptop)
{ return(ptop==NULL);
}
//-----
Void napvao(node*&ptop,intx //Nạp x vào NganXep từ đỉnh
           //của nó
{
node *p;
p = new node;
p->info=x;p->next=ptop;
ptop=p;
};

//-----
//Lay x tu dinh NganXep
int layra(node* &ptop)
{ assert(!empty(ptop));
  int x=ptop->info;
  node* p = ptop;
  ptop=ptop->next;
  delete p;
  return(x);
}
```

```
};

//-----

int ViewTop(node *&ptop)           //Xem phần tử ở đỉnh Stack
{
    assert(!empty(ptop));
    int x=ptop->info;
    return(x);
}

//-----

void deci_bina(int n)           //Hàm chuyển đổi một số
                                //thập phân sang số nhị phân
{
    node *ptop;
    init(ptop);
    int m,t;m=n;
    while(m>0)
    {
        t=m%2;
        napvao(ptop,t);
        m=m/2;
    }
    printf("\n%d chuyen sang he nhi phan la: ",n);
    while(!empty(ptop))
    {
        t=layra(ptop);
        printf("%d",t);
    }
    clear(ptop);
}

//-----

void main()
{
    clrscr();int m;int x;node* ptop, *p;
    clrscr();
```

```

while(true)
{clrscr();
printf("\n 1. Khoi tao ds LIFO ");
printf("\n 2.Nap mot phan tu vao ds LIFO tu
      dinh");
printf("\n 3. Xoa mot phan tu khoi ds LIFO ");
printf("\n 4. Xem phan tu o dinh ds LIFO ");
printf("\n 5. Chuyen so thap phan sang he nhi
      phan.");
printf("\n 0. Ket thuc");
printf("\n Hay go 1, 2, . . . , 0: de chon
      viec!: ");
char chon=getch();
if(chon=='0') break;
printf("\n");
switch(chon)
{
    case '1':init(ptop);break;
    case '2':printf("Nap pt moi vao ds LIFO:");
               scanf("%d",&x);
               napvao(ptop, x);break;
    case '3':printf("\n");
               printf("Da lay ra mot phan tu tu dinh ds
                     LIFO .\n");
               printf("Go Enter quay ve Menu,");
               printf(" roi go 4 de xem phan tu hien tai o
                     dinh ds LIFO .");
               layra(ptop);break;
    case '4':printf("\n");ViewTop(ptop);
               printf("Hien tai phan tu o dinh Stack la:
                     %d",ptop->info);
               break;
}
}

```

```

        case '5':printf("\n Nhập số cần chuyển:");
                    scanf("%d", &m);
                    deci_bina(m);
                }
                printf("\n\nNhấn phím bất kỳ để tiếp tục!");
                getch();
            }
        }
    }
}

```

d<sub>2.9.</sub> Dùng template lớp các đối tượng cài đặt STACK (File: StackList.h, trong C++ chuẩn)

d<sub>2.9.1.</sub> Xây dựng template lớp giao diện với stack

```

#ifndef Stack_list_h
#define Stack_list_h
typedef enum {true, false} bool;
template <class ele_stack> class stack_list
{ public:
stack_list();
bool push (const ele_stack& x)//kiểm tra x nạp vào Stack chưa
bool pop (ele_stack& x)      //kiểm tra x đã bị hủy hay chưa?
bool get_top (ele_stack& x) const; //lấy x từ đỉnh Stack?
bool isEmpty() const;           //Stack có rỗng không?
bool isFull() const;           //Stack có đầy không?
private:
struct node_stack             //khai báo cấu trúc nút trong Stack
{
    ele_stack item;
    node_stack* next;
};
node_stack top;
};

#endif //Stack_list_h

```

d<sub>2.9.2.</sub> Các modul thực thi một vài thao tác cơ bản trên stack cài đặt bởi template

(File: Stacklis.CPP trong C++ chuẩn) (Ở phần đăng ký các hàm tiêu đề (còn gọi là các tiền xử lý), ít nhất bạn cần ghi 2 hàm: Stacklis.h và stdlib.h)

- Khởi tạo Stack

```
template <class ele_stack>
stack_list<ele_stack>:: stack_list()
{
    top=NULL;
}
```

- Kiểm tra xem Stack có đầy không

```
template <class ele_stack>
bool stack_list<ele_stack>::isFull() const
{
    return false; //Hàm trả về false nếu stack chưa đầy, ngược lại là true
} //end isFull
```

- Nạp phần tử mới vào Stack (từ đỉnh của nó)

```
template <class ele_stack>
bool stack_list<ele_stack>::push (const ele_stack& x)
{
    node_stack* old_top; //Đăng ký biến động old_top có kiểu
                          //node_stack (lưu địa chỉ cũ của top)
    bool success; //Đăng ký biến success kiểu bool để
                  //kiểm định sự thành công
    old_top=top; //lưu địa chỉ old_top;
    top=new node_stack; //Cấp phát vùng nhớ cho một nút mới
                        //tại đỉnh stack
    if (top==NULL) //Nếu không đủ bộ nhớ để cấp
    {
        top=old_top; //thì khôi phục lại địa chỉ của top
```

```

    success=false;           //và chỉ thị việc ây thất bại
}
else
{   top->next=old_top;      //Ngược lại tạo liên kết với
                           //nút mới ở old_top
    top->item=x;  success=true; } //nạp x vào nút mới
return success;           //Chỉ thị thành công
} //end push
- Kiểm tra xem Stack có rỗng không
template <class ele_stack>
bool stack_list<ele_stack>::isEmpty() const
{
    return top==NULL;          //Trả về true nếu stack rỗng, ngược
                               //lại trả về false
} // end isEmpty
- Lấy ra (hủy) một phần tử từ đỉnh Stack
template <class ele_stack>
bool stack_list<ele_stack>::pop (ele_stack& x)
{
node_stack* =old_top;
bool success;
if (top=NULL)
    success=false;
else
    {x=top->item; //Ngược lại lưu data ở trường item tại đỉnh vào x
     old_top=top;       //Lưu địa chỉ của top
     top=old_top->next; //Cho con trỏ đỉnh trỏ tới nút kế tiếp
     delete old_top;    //Xóa vùng nhớ đã cấp cho old_top
     success=true;      //Chỉ thị thành công cho việc này
    }
}

```

Nếu stack rỗng thì việc lấy ra một  
phần tử là không thể (thất bại)

```

    return success;
} //end pop

- Cho một phần tử ở đỉnh Stack (nhưng không hủy nó (without popping),
nghĩa là chỉ xem nó))

template <class ele_stack>
bool stack_list<ele_stack>::get_top (ele_stack& x)
    const
{
    bool success;
    if (top=NULL)           //Nếu stack rỗng thì không thực
                            hiện được việc này
        success=false;
    else
        {x=top->item;      //Ngược lại việc cho biết một phần
                            tử ở đỉnh là thành công
        success=true;
    }
    return success;
} //end get_top

```

#### 4.4.2. Kiểu dữ liệu trùu tượng Hàng đợi (Queue ADT)

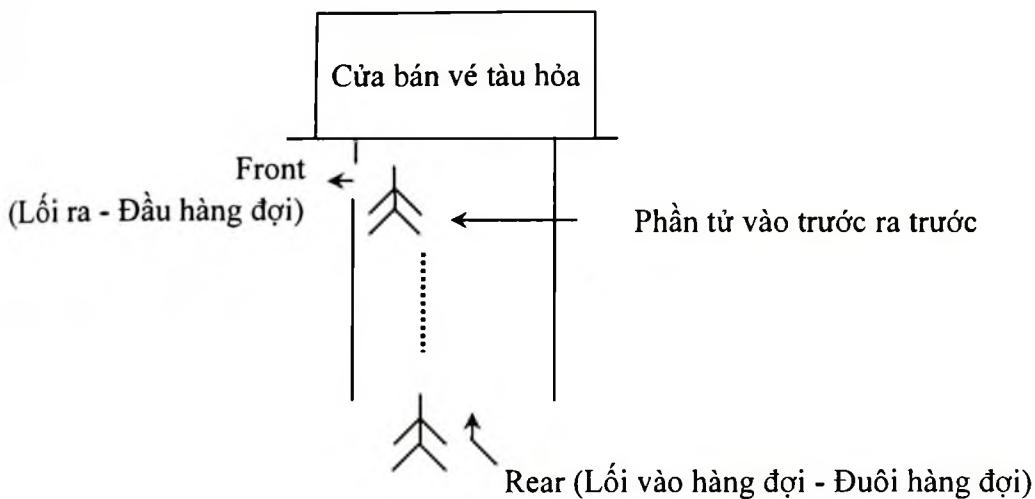
a. **Định nghĩa:** Queue là một danh sách tuyến tính (xem lại mục 4.2) làm việc theo cơ chế “Vào trước ra trước” (First In First Out, viết tắt: FIFO).

Đặc điểm của Queue:

- + Có thể nạp phần tử mới vào Queue bất cứ lúc nào.
- + Song chỉ có phần tử nạp vào đầu tiên mới được lấy ra.
- + Queue có hai lối: Lối vào gọi là Đuôi (Rear) để nạp dữ liệu vào; Lối ra gọi là Đầu hàng đợi (Front) để lấy ra phần tử (vào đầu tiên)



b. Hình ảnh thực tế của Queue là hàng người xếp hàng mua vé tàu hỏa:



Hình 4.5. Hình ảnh thực tế của Hàng đợi

Các thao tác trên Queue:

- Thêm phần tử mới vào Queue, còn gọi là EnQueue (thực hiện ở Rear).
- Lấy ra (hủy) một phần tử của Queue, còn gọi là DeQueue (thực hiện ở Front).
- Kiểm tra xem Queue có rỗng không.
- Xem giá trị của phần tử ở đầu Queue.

#### c. Các công cụ cài đặt Queue

Tương tự danh sách LIFO, Queue có thể cài đặt nhờ mảng 1\_chiều và DSLKD và template. Song với mục đích nghiên cứu DSLKD và vì những ưu điểm của DSLKD đã nêu ở trên so với mảng nên ta sẽ cài đặt Queue bằng DSLKD. Ngoài ra, có thể cài đặt các giao diện của Queue bằng các khuôn mẫu (template, như đã làm với Stack). Còn cài đặt Queue bằng Array sẽ là bài tập tự luyện dành cho độc giả!

#### d. Cài đặt các thao tác cơ bản trên Queue

##### d<sub>1</sub>. Định nghĩa Queue như một DSLKD

```
struct node {int info; node* next; };
struct Queue {node *pfirst,*plast;};
```

##### d<sub>2</sub>. Khởi trị cho Queue

```
void init(Queue &q)
{ q.pfirst=NULL; q.plast=NULL; }
```

d<sub>3</sub>. Kiểm tra xem Queue có rỗng không

```
int empty(Queue &q) {return(q.pfirst==NULL); }
```

d<sub>4</sub>. Nạp một phần tử mới vào Queue (từ Rear)

```
void put(Queue &q, int x) //EnQueue
{node *pp;
 pp = new node; //Tạo nút pp
 pp->info=x; //Nạp x vào trường info của nút
 pp->next=NULL; //pp không trỏ tới đâu cả
 if(q.pfirst==NULL) //Nếu queue rỗng thì
 { q.pfirst=pp; q.plast=pp; return; } //cập nhật địa
                                         chỉ đầu và đuôi queue
 q.plast->next=pp; q.plast=pp;
};
```

d<sub>5</sub>. Lấy ra (xóa) một phần tử của Queue (từ front)

```
int pop(Queue &q) //DeQueue
{ assert(!empty(q)); //Nếu queue không rỗng (thì)
  int x=q.pfirst->info; //Lấy dữ liệu ở trường info tại
                        //nút đầu hàng đợi lưu vào x
  node* p = q.pfirst; //Lưu địa chỉ của queue vào p
  q.pfirst=q.pfirst->next; //Để con trỏ đầu (front) queue
                            //trỏ tới phần tử kế tiếp
  delete p; //Giải phóng vùng nhớ cấp cho p
             //tức đã hủy phần tử ở front)
  if(q.pfirst==NULL) q.plast=NULL; //Nếu queue rỗng thì
                                         //cập nhật địa chỉ cho con trỏ rear
  return(x); //Hàm trả về dữ liệu cần hủy
};
```

d<sub>6</sub>. Xem phần tử ở đầu Queue:

```
int ViewFront (Queue &q)
{ assert (!empty (q)) ; // (Nếu) queue không rỗng (thì)
  int x=q.pfirst->info; //Lấy dữ liệu ở trường info tại nút
                         //đầu hàng đợi lưu vào x
  return (x);           //Hàm trả về dữ liệu cần xem tại
                        //front của queue
};

d7. Xem phần tử ở cuối Queue: (cách giải thích chức năng các lệnh tương
tự phần trên)
```

```
int ViewRear (Queue &q)
{ assert (!empty (q));
  int x=q.plast->info;
  return (x);
};

d8. Xóa toàn bộ Queue
```

```
void clear (Queue &q)
{ node *p,*p1;
  p=q.pfirst; //Cho p trỏ tới front của queue
  while (p!=NULL) //Chừng nào queue không rỗng thì lặp lại các việc:
  { p1=p;          //cắt địa chỉ p vào p1
    p=p->next;    //Xét phần tử kế tiếp
    delete p1; //Giải phóng vùng nhớ cấp cho p1 (tức là xóa nút p1
  }    //và vì việc này lặp lại cho tới khi queue không còn phần tử nào
  q.pfirst=NULL; //tức là xóa toàn bộ queue. Lúc ấy queue
                  //rỗng (q.pfirst=NULL)
}
```

*d9. Cài đặt Queue dùng DSLKD*

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <assert.h>
#define true 1
#define false 0
//-----
struct node {int info; node* next; };
struct Queue {node *pfirst,*plast;};
void init(Queue &q)
{ q.pfirst=NULL;q.plast=NULL; }
//-----
void clear(Queue &q)
{node *p,*p1;
 p=q.pfirst;
 while(p!=NULL)
 {p1=p;
  p=p->next;
  delete p1;
 }
 q.pfirst=NULL;
}
//-----
int empty(Queue &q) {return(q.pfirst==NULL); }
//-----
void put(Queue &q,int x) //Nạp vào Hàng đợi từ dưới
{node *pp;
 pp = new node;

```

```
pp->info=x; pp->next=NULL;
if(q.pfirst==NULL)
{q.pfirst=pp; q.plast=pp; return;}
q.plast->next=pp; q.plast=pp;
};

//-----
int pop(Queue &q)      //Lấy (xóa) x từ đỉnh Hàng đợi
{ assert(!empty(q));
  int x=q.pfirst->info;
  node* p = q.pfirst;
  q.pfirst=q.pfirst->next;
  delete p;
  if(q.pfirst==NULL) q.plast=NULL;
  return(x);
};

//-----
int ViewFront(Queue &q)  //Xem phần tử ở đầu Hàng đợi
{ assert(!empty(q));
  int x=q.pfirst->info;
  return(x);
};

//-----
int ViewRear(Queue &q)   //Xem phần tử x ở dưới Hàng đợi
{ assert(!empty(q));
  int x=q.plast->info;
  return(x);
};

void main()
{clrscr(); node* pfirst; Queue q; int x;
```

```

while(true)
{clrscr();
printf("TRUOC TIEN BAN PHAI GO 1 DE KHOI TAO
QUEUE!!\n\n");
printf("\n 1. Khoi tao QUEUE.");
printf("\n 2. Nap mot phan tu vao QUEUE tu
DUOI.");
printf("\n 3. Lay ra (XOA) mot phan tu tu DINH
cua QUEUE.");
printf("\n 4. Xem phan tu o DINH cua QUEUE.");
printf("\n 5. Xem phan tu o Duoi cua QUEUE.");
printf("\n 0. Ket thuc");
printf("\n Chon mot trong cac viec 1,2,...,0: ");
char chon=getch();
if(chon=='0') break;
printf("\n");
switch(chon)
{case '1':init(q) ;break;
case '2':printf("Nap pt moi vao Queue:");
scanf("%d",&x);
put(q, x); break;
case '3':printf("\n");
printf("Da lay ra mot phan tu tu dinh
Queue.\n");
printf("Go Enter quay ve Menu,");
printf("roi go 4 de xem phan tu hien
tai o dinh Queue.");
pop(q);break;
case '4':printf("\n");
printf("Hien tai phan tu o dinh Queue
la:");
}
}

```

```

        printf("%d", ViewFront(q));
        break;
    case '5':printf("\n");
    printf("Hien tai phan tu o duoi Queue la:");
        printf("%d", ViewRear(q));
        break;
    }
    printf("\n\nNhan phim bat ky de tiep tuc!");
    getch();
}
}

```

*e. Các ứng dụng của Queue:* Queue có một số ứng dụng trong thực tế, ứng dụng hay thấy nhất là Queue dùng để lưu các đỉnh của một đồ thị khi tìm kiếm theo thuật toán BFS trên đồ thị. Ngoài ra Queue còn ứng dụng để tổ chức quản lý các tiến trình của hệ điều hành, tổ chức bộ đệm bàn phím, mô phỏng tiến trình thu phí cầu đường các ôtô...

*f. Cài đặt Queue bằng template các lớp giao diện: (File: Queue.h trong C++ chuẩn)*

```

#ifndef queue_h_;
#define queue_h_;
template <class ele_q>
class queue
{ public:
    queue();
    bool insert (const ele_q &x); //thêm phần tử x vào Queue
    bool remove (ele_q& x);      //xóa phần tử x trong Queue
private:
    struct node_q
    {
        ele_q item;
        node_q *next;
    };
}

```

```

    node_q *front;
    node_q *rear;
    int num_items;
} ;
#endif queue_h_;

```

**g. Các hàm thành viên thực thi các thao tác trên queue cài bởi template  
(File: Queue.CPP trong C++ chuẩn)**

*g<sub>1</sub>. Khởi trị cho queue*

```

template <class ele_q> queue<ele_q>::queue()
{
    num_items=0;
    front=NULL;
    rear=NULL;
}

```

*g<sub>2</sub>. Thêm phần tử mới vào queue (từ đuôi q)*

```

template <class ele_q>
bool queue<ele_q>::insert(const ele_q &x)
{
    if (num_items==0)
        {rear=new node_q;
         if (rear==NULL) return false;
         else front=rear;
        }
    else
        {rear->next=new node_q;
         if (rear->next=NULL) return false;
         else rear=rear->next;
        }
    rear->item=x; num_items++;
    return true;
} //end insert

```

g3. Hủy (lấy ra) một phần tử từ trước của queue

```
template <class ele_q>
bool queue<ele_q>::remove ( ele_q &x)
{
    node_q* old_front;
    if (num_items==0)
        {return false;}
    else
    { old_front=front;
      x=front->item;
      front=front->next;
      old_front->next=NULL;
      delete old_front;
      num_items--;
      return true;
    }
}
```

#### 4.4.3. Kiểu dữ liệu trừu tượng Hàng đợi ưu tiên (Priority Queues ADT - PQ)

##### a. Nhắc lại khái niệm cũ để định nghĩa khái niệm mới PQ

Một ngăn xếp là một “kho chứa” dữ liệu làm việc theo cơ chế LIFO; một hàng đợi là một “kho chứa” dữ liệu làm việc theo cơ chế FIFO, còn:

**Hàng đợi ưu tiên là một “kho chứa” dữ liệu làm việc theo cơ chế ưu tiên B<sub>p</sub>IIFO (Biggest priority Input First Out):** nghĩa là phần tử nào thuộc hàng đợi này có giá trị ưu tiên cao nhất sẽ ra trước (B: Biggest: lớn nhất, p: priority: ưu tiên).

Hàng đợi ưu tiên được ứng dụng rộng rãi trong các hệ điều hành, trong các hệ thống máy tính. Ví dụ trong mạng Lan (mạng cục bộ) của một cơ quan có một máy in phải chia sẻ cho nhiều máy tính trên mạng. Trong trường hợp này các tài liệu cần in của các máy tính khác nhau sẽ được lưu tạm thời **trong một hàng đợi ưu tiên** với độ ưu tiên là kích cỡ của tài liệu (đo bằng KB): tài liệu có kích cỡ nhỏ nhất sẽ được ưu tiên in trước nhất (giá trị ưu tiên lớn nhất).

**b. Cài đặt PQ bởi template với Heap**

Không có “công cụ” nào phù hợp hơn Heap để cài đặt PQ, vì phần tử có giá trị lớn nhất luôn ở gốc (đầu) của Heap do vậy việc chèn hay hủy một phần tử của PQ biểu diễn bởi Heap rất thuận tiện.

*b<sub>1</sub>. Dùng một Heap để xây dựng template lớp giao diện cho PQ*

```
typedef enum{true, false} bool;
template <clas Q>
class prio_queue
{ public:
    prio_queue();
    prio_queue (const prio_queue&); //hàm kiến tạo Queue
    ~prio_queue(); //hàm hủy Queue
    prio_queue& operator=(const prio_queue); //sử dụng
                                                toán tử gán
    int size() const; //kích thước Queue
    bool empty() const; //Queue có rỗng không?
    const Q& top() const; //phần tử ở lối ra Queue
    void push(const Q); //nạp vào Queue
    void pop(); //lấy ra khỏi Queue
protected:
    vector<Q> _; //Ta dùng một vector trong C++ chuẩn để lưu các
                    phần tử Heap
    void HeapDown();
    void HeapUp();
};
```

*b<sub>2</sub>. Các hàm thành viên mô tả các thao tác cơ bản trên PQ*

```
template <clas Q>
prio_queue<Q>::prio_queue(); //Kiến tạo (Constructor)
{
}
template <clas Q>
```

```
prio_queue<Q>:: prio_queue(const prio_queue() &
                           q)): _ (q._)           //Sao chép Constructor
{
}

template <clas Q>                                //Hùy kiến tạo
prio_queue<Q>::~prio_queue()
{
}

template <clas Q>
prio_queue<Q>&prio_queue<Q>::operator=(const
                           prio_queue& q)    //Sử dụng toán tử gán
{
    _=-q._;
}

template <clas Q>
int prio_queue<Q>::size() const    //Trả về kích thước
                                    // (số phần tử) của Heap
{
    return _ .size();
}

template <clas Q>
bool  prio_queue<Q>::empty() const //Kiểm tra xem
                                    //PQ có rỗng không
{
    return _ =empty();
}

template <clas Q>
const Q&  prio_queue<Q>::top() const //Trả về phần tử
                                         //ở đỉnh (front) PQ
{
    return _ =front();
}
```

```

    }

template <clas Q>
void prio_queue<Q>:: pop() //Lấy ra phần tử ở trước của PQ
                           (tức là phần tử ở gốc của Heap)
{
    _front() = _back();
    _pop_back();
    HeapDown(); //để chuyển nó xuống cấp lá của Heap và khôi phục
} //đặc tính của Heap
template <clas Q>
prio_queue<Q>::push(const Q& x)
{
    _push()_back(x); //Nạp phần tử x vào pq từ Rear (từ đuôi)
                      //của PQ
    HeapUp(); //rồi chuyển nó từ mức lá của Heap lên gốc
              //để phục hồi đặc tính của Heap
}

```

### b3. Lưu ý

- *Hàm HeapDown:* Hàm này chuyển phần tử gốc xuống phía dưới đến cấp lá (của cây Heap) theo thứ tự để khôi phục đặc tính Heap. Nhằm mục đích đó, HeapDown phải hoán đổi mỗi phần tử phân bố dọc đường dẫn từ gốc đến lá với con lớn hơn. Do biến điều khiển lặp i gấp đôi lên sau mỗi lần lặp thì toàn bộ tiến trình trên phát sinh số con nhiều hơn  $\lg(n)$  lần lặp. Code mô tả hàm này như sau:

```

template <clas Q>
void prio_queue<Q>::HeapDown()
{
    int n=_size(), j;
    for (i=0; i<n/2; i=j)
        {j=2*i+1; //_[j] và _[j+1] là các con của cha _[i]
         if (_[i]>=_[j]) break;

```

```

        swap(_[i],_[j]);
    }
}

```

- *Hàm HeapUp*: Ngược lại với hàm trên, HeapUp chuyển phần tử lá cuối Heap lên gốc để phục hồi đặc tính của Heap. Nó thực hiện việc ấy bằng cách hoán đổi mỗi phần tử con dọc theo đường dẫn từ gốc đến lá với Cha nhỏ hơn. Do biến điều khiển lặp  $j$  giảm một nửa sau mỗi lần lặp nên toàn bộ tiến trình cần không hơn  $\lg(n)$  lần lặp. Code mô tả hàm này cho dưới đây:

```

template <class Q>
void prio_queue<Q>::HeapUp()
{ int n=_size(), i;
  for (int j=n-1; j>0; j=i)
  { i=(j-1)/2; //_[i] là cha của _[j]
    if(_[j]<=_[i]) break;
    swap(_[j],_[i]);
  }
}

```

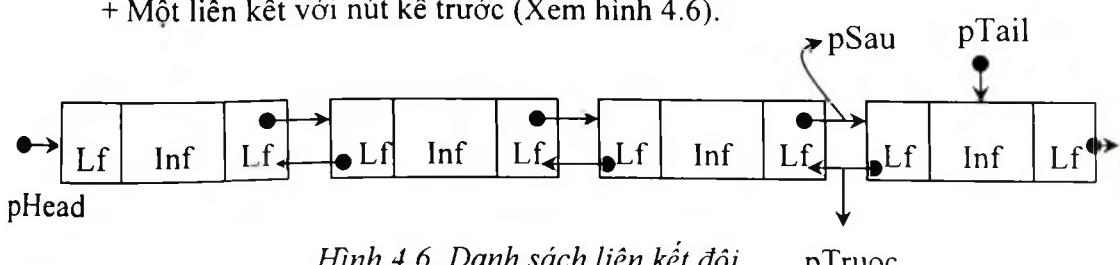
- Việc thực thi một PQ nhờ dùng cấu trúc dữ liệu Heap rất hiệu quả. Cả hàm push() và pop() đều chạy trong thời gian  $O(\lg(n))$ .

## 4.5. KIẾU DỮ LIỆU TRỪU TƯỢNG DANH SÁCH LIÊN KẾT ĐÔI (DOUBLE LINKED LIST ADT)

### 4.5.1. Định nghĩa

*Danh sách liên kết đôi (DSLK\_ĐÔI)* là DSLK mà mỗi nút (node) của nó (trừ nút đầu và nút cuối) có 2 liên kết (2 móc nối):

- + Một liên kết với nút đứng kề sau;
- + Một liên kết với nút kề trước (Xem hình 4.6).



Hình 4.6. Danh sách liên kết đôi

- Lf: Linked Field: Trường liên kết để lưu địa chỉ của nút đứng kề trước hoặc kề sau nút hiện hành.

- Inf: Information Field: Trường thông tin để lưu dữ liệu.

- Mô tả định nghĩa DSLK\_ĐÔI bằng C/C<sup>++</sup>

```
typedef struct Lk_Doi           //Định nghĩa các nút
{
    DataType Info;
    struct Lk_Doi * pTruoc; //Con trỏ liên kết với nút kề trước
    struct Lk_Doi * pSau;  //Con trỏ liên kết với nút kề sau
} DNut;

typedef struct Ds_Lk_Doi //Định nghĩa DSLK_ĐÔI
{
    DNut* pHead;           //Con trỏ đầu danh sách
    DNut* pTail;          //con trỏ cuối danh sách
} D_Ds;
```

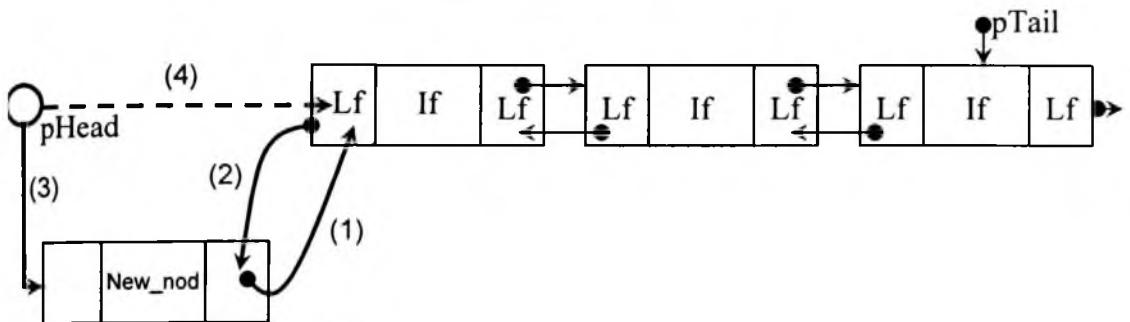
#### **4.5.2. Các thao tác cơ bản trên DSLK\_ĐÔI**

##### *a. Tạo nút mới cho DSLK\_ĐÔI*

```
DNut* TaoNut (DataType x)
{
    DNut* p;
    p=new DNut;
    if (p==NULL)
        {cout<<"Khong du bo nho";
         exit(1);
        }
    p->Info=x;
    p->pTruoc=NULL;
    p->pSau=NULL;
    return p;
}
```

### b. Chèn nút mới vào đầu DSLK\_ĐÔI

Xem sơ đồ sau để thấy thực chất của thao tác này gồm những việc gì!



Phản tử New\_nod cần chèn

- (1): Tạo liên kết xuôi giữa New\_nod và nút đầu ds
- (2): Tạo liên kết ngược giữa nút đầu ds với New\_nod
- (3): Tạo liên kết giữa pHead và New
- (4): Xong 3 việc trên thì liên kết giữa pHead và nút đầu (tự động) bị “cắt” (hủy)

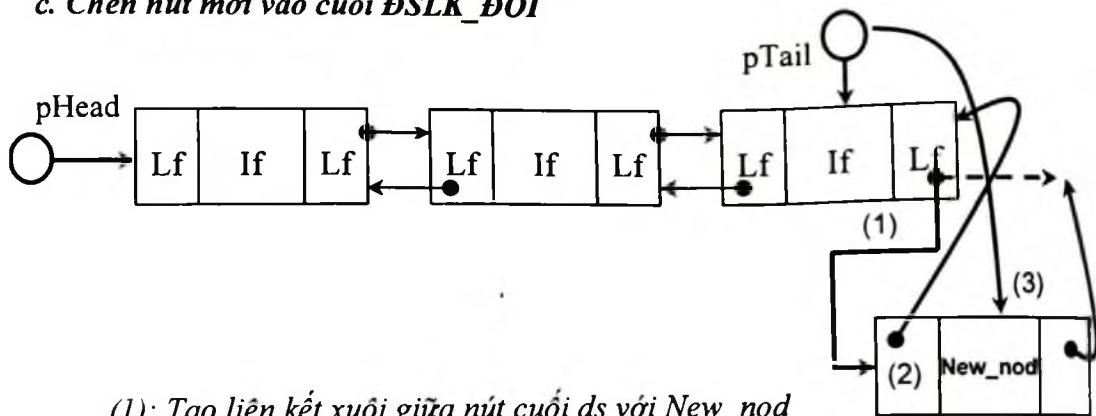
Hình 4.7. Mô tả thao tác chèn nút New\_nod vào đầu ds

```

DNUT* HeadInser(D_Ds &L, DataType x)
{
    DNUT* New_nod=TaoNut(x);
    if (New_nod==NULL)    return NULL;
    if (L.pHead==NULL)
    {
        L.pHead=New_nod;
        L.pTail=L.pHead;
    }
    else
    {
        New_nod->pSaut=L.pHead;           //thao tác (1)
        L.pHead->pTruoc=New_nod;         //thao tác (2)
        L.pHead=New_nod;                 //thao tác (3)
    }
    return New_nod;
}

```

c. Chèn nút mới vào cuối ĐSLK\_ĐÔI



(1): Tạo liên kết xuôi giữa nút cuối ds với New\_nod

(2): Tạo liên kết ngược giữa nút New\_nod với nút cuối ds

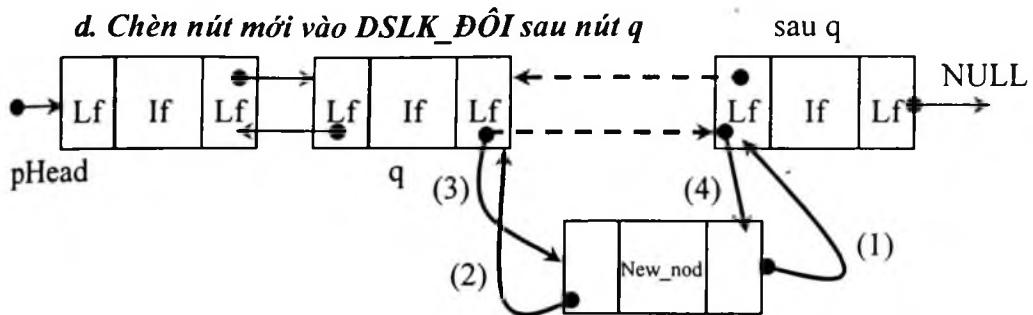
(3): Tạo liên kết giữa Nút New\_nod với con trỏ pTail (tức là cho ptail trỏ tới New\_nod)

Hình 4.8. Mô tả thao tác chèn New vào cuối ds

```

DNUT* TailInser(D_Ds &L, DataType x)
{
    DNUT* New_nod=TaoNut(x);
    if (New_nod==NULL)      return NULL;
    if (L.pHead==NULL)
    {
        L.pHead=New_nod
        L.pTail=L.pHead;
    }
    else
    {
        L.pTail->pSau=New_nod;           // (1)
        New_nod->pTruoc=L.pTail;       // (2)
        L.pTail=New_nod;                // (3)
    }
    return New_nod;
}

```



(1): Tạo liên kết sau giữa New\_nod với nút kè sau q.

(2): Tạo liên kết trước giữa nút New\_nod với q.

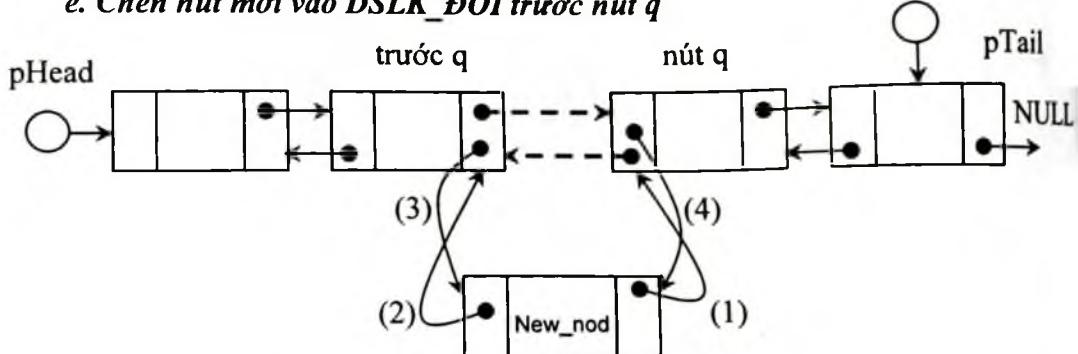
(3): Tạo liên kết sau giữa q với New\_nod.

(4): Tạo liên kết trước giữa nút kè sau q với New\_nod.

Hình 4.9. Mô tả thao tác chèn nút New\_nod vào sau nút q

```
void AfterInser(D_Ds &L, DNUT *q, DataType x)
{
    DNUT* p=q->pSau;
    DNUT* New_nod=TaoNut(x);
    if (New_nod=NULL) return NULL;
    if (q!=NULL)
    {
        New_nod->pSau=p; // (1)
        New_nod->pTruoc=q; // (2)
        q->pSau=New_nod; // (3)
        if (p!=NULL)
            p->pTruoc=New_nod; // (4)
        if (q==L.pTail)
    }           L.pTail=New_nod;
    else
        HeadInser(L, x); //Chèn vào đầu danh sách
}
```

## e. Chèn nút mới vào DSLK\_ĐÔI trước nút q



Hình 4.10. Mô tả thao tác chèn New\_nod vào trước nút q

```

void BefoInser(D_Ds &L, DNUT *q, DataType x)
{
    DNUT* p=q->pSau;
    DNUT* New_nod=TaoNut(x);
    if (New_nod==NULL)      return NULL;
    if (q!=NULL)
    {
        New_nod->pSau=q;                      //(1)
        New_nod->pTruoc=p;                     //(2)
        q->pTruoc=New_nod;                   //(3)
        if (p!=NULL)
            p->pSau=New_nod;
            p->pSau=New_nod;                  //(4)
        if (q==L.pHead)
            L.pHead=New_nod;
    }
    else    HeadInser(L, x);                //Chèn vào đầu ds
}

```



```

L.pTail=L.pTail->pTruoc //Cho L.pTail trỏ tới nút kề trước
L.pTail->pSau=NULL;      //Con trỏ cuối ds trỏ tới pSau
                           rỗng (không trỏ tới đâu)
delete p;                  //Xóa vùng nhớ cấp cho p (tức xóa x)
if (L.pHead==NULL)
    L.pTail=NULL;
else L.pHead->pTruoc=NULL;
}
return x;
}

```

*f3. Xóa một nút có khóa k tùy ý*

```

void Del_Any_Nod(D_Ds &L, DataType k)
{DNUT *p=L.pHead;
 DNUT *q;
 p=L.pHead; q=NULL;
 while (p!=NULL) && p->Info!=k) //Chừng nào danh sách
                                   chưa rỗng thì lặp lại các thao tác sau:
 {p=q;
  p=p->pSau;
 }
 if(p==NULL) cout<<"Khong co khoa k trong Ds.";
 q=pp->pTruoc;
 if (q!=NULL)
 {p=q->pSau;
  if (p!=NULL)
   {q->pSau=p->pSau;
    if (p==L.pTail)
     L.pTail=q;
    else p->pSau->pTruoc=q;
   }
 }

```

```

    }

else
{   L.pHead=p->pSau;
    if (L.pHead==NULL)
        L.pTail=NULL;
    else
        L.pHead->pTruoc=NULL;
}
delete p;
return L;
}

```

#### *f<sub>4</sub>. Sắp xếp DSLK\_ĐÔI*

Như các phần trên đã nói, để sắp xếp các DSLK thì hiệu quả hơn cả là thuật toán Quick Sort. Vì vậy dưới đây ta sẽ dùng thuật toán ấy để sắp xếp DSLK\_ĐÔI.

```

void DDS_Qsort(D_Ds & L)
{ DNUT *p, *m;                      //m: lưu địa chỉ của nút mốc (mốc
                                         //dùng để phân hoạch dãy L)

D_Ds L1, L2;
if(L.pHead==L.pTail) return; //Nếu danh sách đã có thứ tự
                            //thì trả quyền điều khiển cho main()

L1.pHead==L1.pTail=NULL; // Khởi trị cho danh sách con L1
L2.pHead==L2.pTail=NULL; // Khởi trị cho danh sách con L2
m=L.pHead;
L.pHead=m->pSau;
while(L.pHead!=NULL)
{
    p=L.pHead;
    L.pHead=p->pSau;
    p->pSau=NULL;
}

```

```

        if (p->Info<=m->Info)
            TailInser(L1, p);
        else
            TailInser(L2, p);
    }

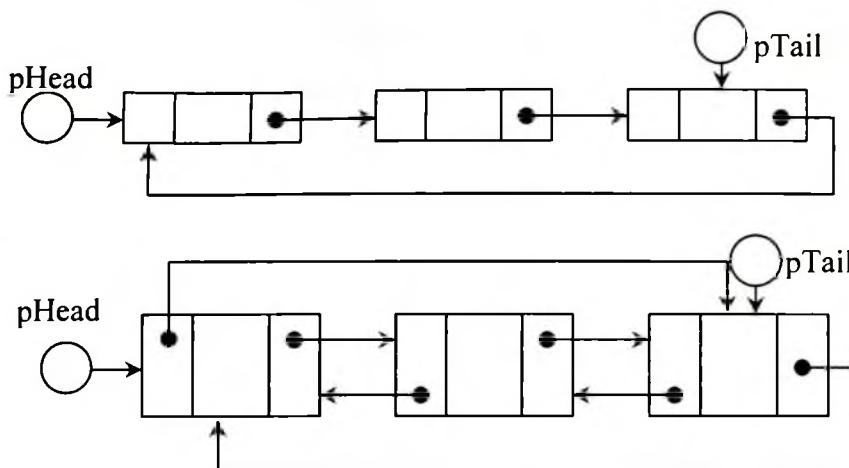
    DDS_Qsort((L1));           //Gọi đệ quy để xếp danh sách con L1
    DDS_Qsort((L2));           //Gọi đệ quy để xếp danh sách con L2
    //Dưới đây là thao tác ghép nối L1, m, L2 để được L sắp xếp có thứ tự
    if (L1.pHead!=NULL)
    {
        L.pHead=L1.pHead
        L1.pTail->pSau=m;
        m->pTruoc=L1.pTail
    }
    else
    {
        L.pHead=m;
        m->pSau=L2;
        if (L2.pHead!=NULL)
        {
            L.pTail=L2.pTail;
            L2->pHead->pTruoc=m;
        }
        else
            L.pTail=m;
    }
}

```

## **4.6. KIẾU DỮ LIỆU TRƯỞU TỰNG DANH SÁCH LIÊN KẾT VÒNG**

### **4.6.1. Định nghĩa danh sách liên kết vòng - DSLKV (Circular Linked List-CLL-)**

DSLKv và DSLK\_ĐÔI có con trỏ cuối các danh sách này lại trỏ tới nút đầu danh sách khác, chứ không phải là không trỏ tới đâu (NULL).



Hình 4.11. Danh sách liên kết vòng

#### 4.6.2. Các thao tác cơ bản trên DSLKV

##### a. Tìm một nút có khóa k trên DSLKV

Như từ định nghĩa và hai sơ đồ trên ta thấy DSLKV không có nút đầu tường minh. Tuy nhiên để tiện cho việc rà soát DSLKV, chúng ta có quyền đánh dấu một nút bất kỳ làm nút đầu của DSLKV.

```

DNUT* Find(D_Ds & L, DataType k)
{
    DNUT *p;
    p=L.pHead;
    do
    {
        if(p->Info==k)    return p;
        p=p->pSau;
    }
    while (p!=L.pHead);
    return p;
}
  
```

##### b. Chèn nút mới vào đầu DSLKV

```

void HeadInser( D_Ds &L, DNUT *New_nod)
{
    if(L.pHead==NULL)
  
```

```

    {
        L.pHead=L.pTail=New_nod;
        L.pTail->pSau=L.pHead;
    }
    else
    {
        New_nod->pSau=L.pHead;
        L.pTail->pSau=New_nod;
        L.pHead=New_nod;
    }
}

```

**c. Chèn nút mới vào cuối DSLKV**

```

void TailInser(D_Ds &L, DNUT *New_nod)
{
    if(L.pHead==NULL)
    {
        L.pHead=L.pTail=New_nod;
        L.pTail->pSau=L.pHead;
    }
    else
    {
        New_nod->pSau=L.pHead;
        L.pTail->pSau=New_nod;
        L.pTail=New_nod;
    }
}

```

**d. Chèn nút mới vào sau nút q trong DSLKV**

```

void AfterInser(D_Ds &L, DNUT *q, DNUT *New_nod)
{
    if(L.pHead==NULL)
    {

```

```

        L.pHead=L.pTail=New_nod;
        L.pTail->pSau=L.pHead;
    }
else
{ New_nod->pSau=q->pSau;
    q->pSau=New_nod;
    if(q==L.pTail)
        L.pTail=New_nod;
    }
}

```

**e. Xóa nút ở đầu DSLKV**

```

void HeadDel(D_Ds &L)
{DNUT *p=L.pHead
if(p==NULL) return;
if(L.pHead==L.pTail)
    L.pHead=L.pTail=NULL;
else
{ L.pHead=p->pSau;
    if(p==L.pTail)
        L.pTail->pSau=L.pHead;
    }
delete p;
}

```

**f. Xóa nút đứng sau nút q trong DSLKV**

```

void After_q_Del(D_Ds &L, DNUT *q);
{DNUT *p;
if(q!=NULL)
{p=q->pSau;
if(p==q)
    L.pHead=L.pTail=NULL;
}

```

```

    else
    {
        q->pSau=p->pSau;
        if (p==L.pTail)
            L.pTail=q;
    }
    delete p;
}
}

```

## BÀI TẬP CHƯƠNG 4

4.1. Dùng mảng cài đặt:

- a. Stack.
- b. Queue.

4.2. Xác định xem từng biểu thức sau đây có là Postfix không?

- a.  $xy+z+=xyz++$
- b.  $xy+z-=xyz-$
- c.  $xy-z+=xyz+-$
- d.  $xy-z-=xyz-$

4.3. Viết template sau đây:

```
template <class T> void reverse(Stack<T>&); //đảo ngược nội dung của stack
```

4.4. Vẽ sơ đồ dò vết cho biết nội dung của stack với mỗi lệnh gọi sau:

```

stack <char> s;
s.push('A');
s.push(( 'B' ));
s.push(( 'C' ));
s.pop();
s.pop();
s.push('D');
s.push('E');

```

```
s.push('F');
s.pop();
s.push('G');
s.pop();
s.pop();
s.pop();
```

4.5. Viết hàm thành phần sau đây cho stack:

T& bottom(); //Trả về phần tử ở đáy stack

4.6. Viết code cho một ngăn xếp thực thi hàm sau:

bool isPalindrom(string s) //Kiểm tra xem xâu đã cho có là Palindrome không?

4.7. Viết toán tử đẳng thức sau đây cho các stack:

bool Stack::operator==(const Stack&);

4.8. Viết toán tử gán sau đây cho Stack:

bool Stack::operator+=(const Stack&);

4.9. Viết các hàm thành viên sau cho Stack:

a-void Stack::clear(); //Làm rỗng Stack

b-void Stack::popBottom(); //Đẩy phần tử ở đáy Stack ra khỏi nó

4.10. Viết code thực thi phép tính biểu thức RPN.

4.11. Chuyển từng Postfix sau đây về dạng Infix và tính nó:

a. \*-9+2 3-7 1

b. /+\* 3 8 4 + 6-3 2

4.12. Dùng Queue mô phỏng tiến trình các xe hơi nộp phí đường bộ tại trạm thu phí có 4 ki-ốt thu phí. Đơn vị thời gian được tính bằng 1/10 giây. Thời gian giữa các lần xe đến và thời gian dừng xe để nộp lộ phí là các biến ngẫu nhiên được phân bố tăng dần. Với giả thiết như vậy ta thiết đặt thời gian trung bình giữa các lần xe đến là 2giây (=20 lần của 1/10 giây).

4.13. Dò vết sau khi thực hiện các lời gọi trong queue dưới đây;

```
queue<char> s;
q.push('A');
q.push('B');
```

```

q.push('C');
q.pop();
q.pop();
q.push('D');
q.push('E');
q.push('F');
q.pop();
q.push('G');
q.pop();
q.pop();
q.pop();

```

4.14. Viết template sau đây:

```
template <class T> void reverse(queue<T>&); //đảo ngược nội dung của queue
```

4.15. Viết các toán tử sau cho queue:

- bool queue::operator==(const queue&); //toán tử ==
- bool queue::operator+=( const queue &) //toán tử gán, nạp phần tử mới vào queue

4.16. Sử dụng code của bài 4.12 tính và in từng số liệu thống kê đây lên màn hình. Kiểm thử nó trên một lần chạy kéo dài trong 1 giờ (36000 lần của 1/10 giây):

- Số xe ôtô trung bình trong hàng đợi đóng lộ phí;
- Thời gian đợi trung bình trong hàng đợi;
- Thời gian thu lộ phí trung bình của tất cả các ôtô nộp lộ phí;
- Thời gian thu lộ phí trung bình của mỗi ki-ốt thu lộ phí;
- Thời gian phần trăm nhàn rỗi của mỗi ki-ốt thu lộ phí;

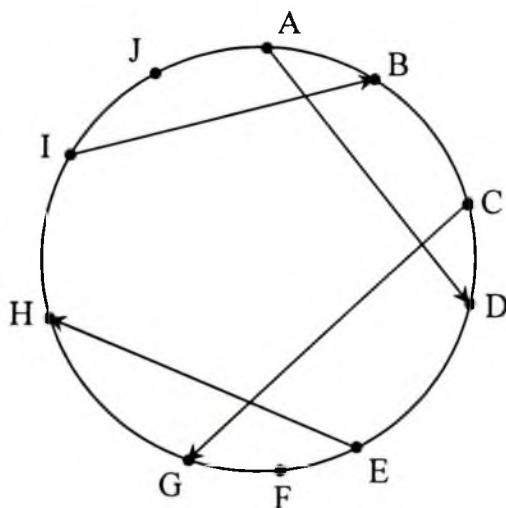
4.17. (*Một áp dụng của Hàng đợi ưu tiên*) Một mạng LAN (mạng cục bộ) của một công ty dùng chung một máy in cho tất cả các PC trong mạng này. Khi đó các tài liệu cần in được lưu trên hàng đợi ưu tiên và công ty quy định tài liệu có kích cỡ (đo bằng Kilô Byte-KB) nhỏ hơn sẽ được in trước so với các tài liệu lớn hơn. Hãy viết một chương trình dùng Priority Queue mô phỏng tiến trình đó.

4.18. Một đối tượng List khác một đối tượng Array như thế nào?

4.19. Một danh sách tự tổ chức (SelfOrganizingList) là gì? Ích lợi của nó? Viết một mẫu (template) các lớp con cho SelfOrganizingList.

4.20. (**Bài toán Josephus**) Bài toán này dựa vào lời kể của sử gia Joseph Ben Marthias (“Josephus”) về một kiểu tự sát của 40 người lính và ông khi bị quân La Mã hùng mạnh bao vây vào năm 67 sau công nguyên. Khi ấy Joseph đề nghị mỗi người giết mỗi người đứng kè bên mình và cố gắng xoay sở để trở thành người sống sót sau cùng trong số các người lính ấy. Do sự khôn khéo, ông ta đã sống sót và kể lại sự kiện này. Bài toán Josephus mô phỏng sự kiện đó.

Gọi  $n$  là số người đứng thành một vòng tròn và họ đều quay mặt vào trong, gọi  $k$  là một “số bỏ qua” được đồng ý trước, gọi  $a$  là người bắt đầu thực hiện việc nêu trên. Vào mỗi lần lặp lại,  $x$  sẽ giết người thứ  $k$  đứng ở bên trái mình, trong đó  $x$  bắt đầu với  $A$  và được xác lập lại để trở thành người đứng bên trái của người bị giết mỗi lần. Ví dụ với  $n=10$  và  $k=3$  thì  $A$  giết  $D$ ,  $E$  giết  $H$ ,  $I$  giết  $B$ ,  $C$  giết  $G$  (xem sơ đồ hình 4.12). Hãy xây dựng class thực thi bài toán Joseph.



*Hình 4. 12. Minh họa tiến trình của bài toán Joseph*

4.21. Dùng danh sách liên kết đơn và class khai báo cấu trúc của đa thức và viết các hàm mô tả những thao tác cơ bản trên hai đa thức.

4.22. Viết code với danh sách liên kết đơn để lưu và thực thi một số thao tác cơ bản trên dãy số nguyên. Chương trình gồm một thực đơn sau:

1. Khởi tạo danh sách liên kết đơn;

2. Nhập dữ liệu vào danh sách liên kết đơn sao cho người dùng có thể tùy chọn một trong hai cách sau:

- 2.1. Nhập bằng tay;
- 2.2. Nhập tự động
3. Xóa 1 phần tử trong danh sách liên kết đơn
4. Tìm 1 phần tử có khóa k trong danh sách đơn
5. Sắp xếp danh sách liên kết đơn
6. Duyệt danh sách

Thiết kế các hàm này trên danh sách liên kết đơn không dễ và quen thuộc như trên mảng. Công việc này là không quen thuộc và khó với nhiều sinh viên, đặc biệt là sắp xếp danh sách liên kết đơn!

4.23. Viết code với danh sách liên kết đơn quản lý một số thông tin cho việc tuyển sinh: họ tên thí sinh, mã thí sinh, tên ngành, các điểm thi: toán, lý, hóa, điểm tổng. Thực đơn của chương trình gồm các mục dưới đây:

- F1. Khởi tạo danh sách
- F2. Nhập dữ liệu
- F3. Xem danh sách thí sinh có điểm tổng  $\geq 18$
- F4. Xếp danh sách theo điểm tổng tăng
- F5. Xem toàn bộ danh sách
- F6. Tìm kiếm theo mã số
- F10. Thoát

**Lưu ý:** a. Về tổ chức thực đơn, bài 4.23 khác bài 4.22 ở chỗ: Yêu cầu người viết code phải dùng **các phím chức năng** để chỉ thị các khoản mục thuộc thực đơn như trên đã liệt kê. Điều này không dễ với khá nhiều sinh viên.

b. Yêu cầu bạn phải khai báo các nguyên mẫu của các hàm trước tiên (Đó là dạng chuẩn, hãy so sánh với code của bài 4.21) để tự rút ra hiểu biết về cách tổ chức chương trình.

c. Kết quả của chức năng F3 và F5 in theo form sau:

Ho ten	Ma so	Dtoan	DLy	DHoa	Dtong
Yen	901	5	8	7	20

**GỢI Ý HOẶC ĐÁP ÁN**

4.1. a. Cài đặt Stack bởi mảng:

```
#include<iostream.h>
#include<conio.h>
#define ln 10
struct nx
{ int a[ln];
  int top;
};
struct nx s;
void napvao(int n);
void duyet();
int layra();
void main()
{ int c,n; clrscr();
  s.top=0;
  do
  { cout<<"\n 1. Nap vao.";
    cout<<"\n 2. Lay ra.";
    cout<<"\n 3. Duyet.";
    cout<<"\n 4. Thoat.";
    cout<<"\n 5. Chon(1..4). Go 1 nhieu lan de
nap nhieu pt!:";
    cin>>c;
    if (c==1) { cout<<" Nap n="; cin>>n;
napvao(n);}
    if (c==2)
      { n=layra();
        if (n!=0)
          cout<<n;
      }
  }
```

```

        if (c==3) duyet();
    } while (c!=4);
}

void napvao(int n)
{
    if (s.top>=ln) {cout<<"\n Stack day,
top=<<s.top;}
    else {s.top++; s.a[s.top]=n;}
}

void duyet()
{
    int i;
    if (s.top==0) cout<<"\n Stack rong";
    else { for(i=1; i<=s.top; i++)
            cout<<"\n Phan tu thu "<<i<<" la:"<<s.a[i];
    }
    cout<<"\n Do la phan tu vao sau cung";
}

int layra()
{
    int n;
    if (s.top==0){ cout<<"\n Stack rong."; return 0;}
    else { cout<<"\n Phan tu vao sau cung duoc lay ra
truoc tien la:";
            n=s.a[s.top]; s.top--; return n;}
}

```

b. Làm tương tự!

4.12. Code sử dụng 3 lớp:

- Lớp Exp chứa các đối tượng tạo các số ngẫu nhiên phân bố tăng dần;
- Lớp Car có các đối tượng biểu thị các ô tô;
- Lớp Kiot gồm các đối tượng biểu thị cho 4 kiot thu phí.

```
const int Kiot=4;
const int sove=600;
const float av_arri=20; //thời gian đến trung bình của ô tô
const float dv=80          //trung bình mất 8 giây cho 1 dịch vụ
class Exp: public Random
{ public:
    Exp(float mean):_mean(mean) {}
    float time() {return-_mean*log(1-real());}
private:
    float _mean;
};

class Car
{ friend class Kiot;
    friend ostream& operator<<(ostream& ostr, const
Car& car)
    { ostr<<"#"<<car._id<<"}"<<car.server
        <<","<<car._service<<","<<car._server
        <<","<<car._exit<<"}";
    return ostr;
}
static int _cout;
public:
    car(int time=0):_id(_cout++ , _arrive(time()),
        _sevice(0), _remaining(0.0), _exit(0) {}
    int id() { return _id};
    boolfinished() {return _remaining<=0.0;}
protected:
    int _id;
    int _arrive;
    int _sevice;
```

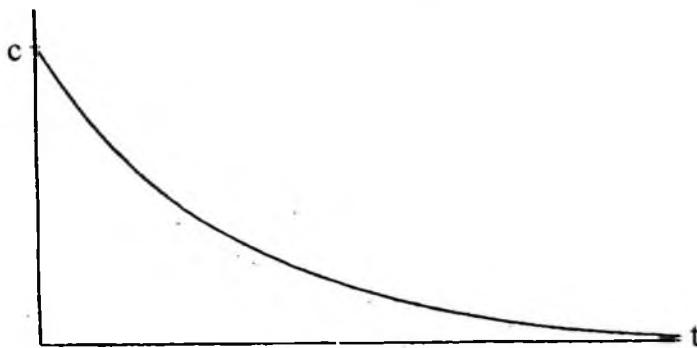
```
        float _remaining;
int _exit;
char _server;
};

class kiot
{ static Exp _service;
public:
    Kiot():_service(false), _p(0) {}
    void setId(char ch) {_id = ch;}
    bool isServing() {return _serving;}
    Car& car() {return *_p;}
    void enter(Car* p, int time)
    { _serving=true;
      _p=p;
      p->_service=time;
      p->_remaining=_service.time;
      p->server=_id;
    }
    void serviceCar(){ _p->_remaining-=1.0;}
    void vacate(int time)
    { _serving=false;
      _p->_exit=time;
    }
private:
    char _id;
    bool _serving;
    Car* _p;
};
int Car::_count=0;
Exp kiot::_service(ST);
```

```
int main()
{ vector<Kiot> kiot(kios);
  for(int i=0; i<kios; i++)
    kiot[i].setId('A'+i);
  queue<Car*> line;
  Exp arrival(IAT);
  float timeToNextArrival=0.0;
  for(intt=0; t<sove; t++, timeToNextArrival-=1.0)
  { if(timeToNextArrival<=0.0)
    { Car* p=new Car(t);
      line.push(p);
      timeToNextArrival=arrival.time();
    }
    for(int i=0; i<Kiot; i++)
      if (kiot[i].isServing())
        { Car& =kiot[i].car();
          koit[i].serviceCar();
          if(car.finished())
            { kiot[i].vacate(t);
              cout<<car<<"\n";
            }
        }
    else if (!line.empty())
      { Car* p=line.front();
        kiot[i].enter(p, t) ;
        line.pop();
      }
  }
}
```

Phân tích code:

- + Lớp Exp là lớp con của lớp Random.
- + Hàm random::real() sản sinh các số x phân bố đều trong khoảng  $0 \leq x \leq 1$ .
- + Hàm Exp::time() tạo các số t phân bố tăng dần với giá trị trung bình \_mean. Hàm mật độ tăng dần có dạng như ở hình 4.13.



Hình 4.13. Hàm mật độ tăng dần

Giá trị c là nghịch đảo của giá trị trung bình. Dạng phân bố này là mô hình toán học chuẩn để mô phỏng các sự kiện xảy ra vào các thời điểm ngẫu nhiên.

+ Lớp Car tạo các đối tượng biểu thị cho các ô tô tại trạm nộp lộ phí. Mỗi đối tượng car có một số nhận dạng \_id (nói gọn là mã số). Thời gian \_arrive (tính bằng 1/10 giây) khi ô tô tới trạm thu phí cầu đường (trạm thu lộ phí trong bài toán của chúng ta có 4 kiot thu phí). Thời gian \_service khi một ô tô đến tại phòng (kiot) thu phí cầu đường. Thời gian \_exit khi một ô tô rời khỏi phòng thu lộ phí. Ký tự \_server nhận biết kiot thu lộ phí.

+ Số nguyên \_remaining để đếm ngược mỗi lần 1/10 giây đến khi kiot kế tiếp đã thu xong lộ phí của một ô tô. Trong lớp có sử dụng một biến đếm tĩnh \_count để tạo các số nhận dạng các ô tô (car #1, car #2, ...).

+ Lớp bạn Kiot cụ thể hóa các đối tượng biểu thị các kiot thu lộ phí. Chương trình chính sử dụng một vector (mảng, trong C++ chuẩn là vector) 4 đối tượng Kiot. Mỗi đối tượng lưu trữ một ký tự nhận dạng \_id (phòng A, phòng B, ...). Một cờ (dấu hiệu) \_serving để xác định xem Kiot thu phí hiện hành có đang thu lộ phí một xe hay không và một pointer \_p trỏ tới xe hơi mà nó có đang bị thu phí hay không.

Lớp có 3 hàm thay đổi:

- enter(): được gọi khi một xe hơi đã đến một Kiot thu lô phí.
- serviceCar(): được gọi một lần vào mỗi 1/10 giây mà trong suốt thời điểm ấy Kiot đang thu lô phí xe này.
- vector() được gọi khi xe rời khỏi Kiot thu lô phí.
- + Đầu tiên main() khởi trị cho vector của 4 đối tượng Kiot.
- + Hàng đợi line gồm các pointer Car.
- + Bộ tạo số ngẫu nhiên Exp arival sẽ tạo thời gian giữa các lần đến của các xe hơi và một số nguyên timeToNextArrival ghi nhớ khoảng thời gian bao lâu trước khi xe hơi kế tiếp đi đến. Sau đó nó bắt đầu vòng lặp định thời gian chính cứ lặp lại mỗi lần cho từng 1/10 giây. Nếu đến lúc một xe hơi kế tiếp, nó tạo 1/10 giây và chèn giá trị này vào phía đuôi hàng đợi line. Sau đó nó kiểm tra mỗi trong 4 Kiot thu phí. Nếu 1 Kiot đang thu lô phí thì hàm serviceCar() (ứng với Kiot đang thu phí ấy) được gọi để cập nhật các dữ liệu về xe hơi (đã bị thu phí) và sau đó kiểm tra xem đã đến lúc xe phải rời khỏi Kiot hay không.

Nếu một Kiot không thu phí một xe nào và đang có nhiều xe hơi đang chờ trong hàng đợi line thì nó di chuyển xe hơi đứng phia trước đến tại kiot thu phí đó.

+ Kết quả gồm một dòng cho mỗi xe hơi rời khỏi Kiot thu lô phí... Nó hiển thị mã số của xe hơi, thời gian xe tới trạm, thời gian xe đến tại một Kiot thu lô phí, tên của Kiot đó và thời gian xe rời khỏi Kiot này. Các hàng đợi in theo thứ tự mà các xe đã rời khỏi các Kiot thu lô phí theo thứ tự đó.

#3 (38, 38, C, 123) /Ký tự “#” đọc là “số”

#2 (24, 24, B, 143)

#1 (0, 0, A, 195)

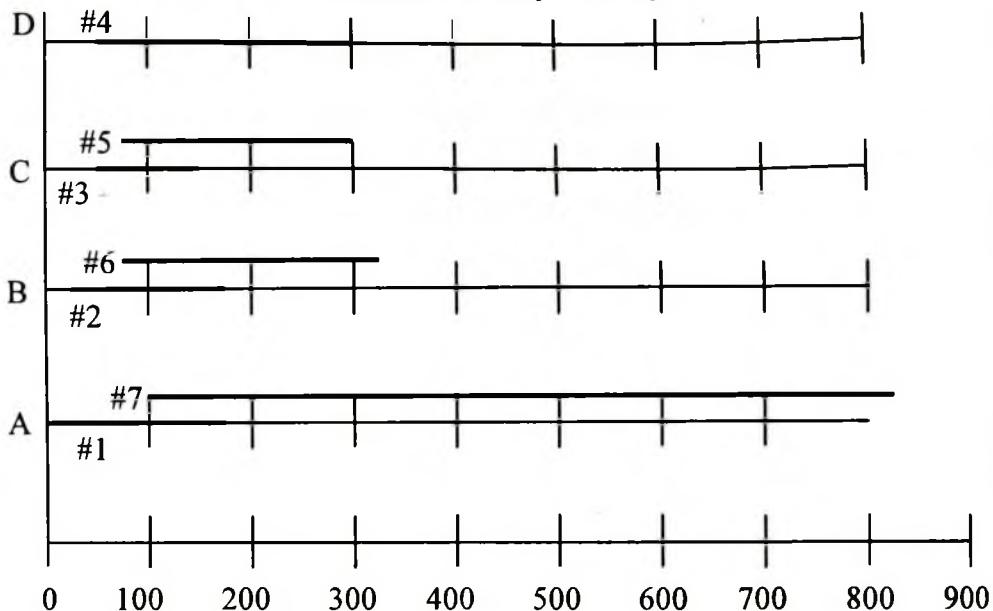
#5 (73, 124, C, 247)

#4 (48, 48, D, 254)

#6 (88, 144, B, 262)

#7 (102, 196, A, 824)

Dưới đây là biểu đồ của tiến trình chạy trên đây:



- 4.19. a. Một SelfOrganizingList là một danh sách mà các phần tử của nó tự động dịch chuyển về phía đầu danh sách vào bất cứ lúc nào nó được truy cập.  
 b. Tính tự tổ chức này cải thiện hiệu quả của việc tìm kiếm trên danh sách.  
 c. Dưới đây là một hàm template class cho các SelfOrganizingList:

```
template<class T>
class SelfOrganizingList:public list<T>
{
public:
    bool contains(const T&)
;

    template<class T>
    bool SelfOrganizingList<T>::contains(const
        T& t)
    {
        list<T>::iterator it=find(begin(),end(), t);
        if(it==end()) return false;
        if(it!=begin())
        {
            erase(it);
            push_front(t);
        }
    }
};
```

```
    }

    return true;
}
```

#### 4.20. (Bài toán Josephus)

```
class List           //List này có trong thư viện của C++ chuẩn
{ protected:
    struct Node
    { Node(const char& ch=0) : _ch(ch) {}           //Constructor
        char _ch;
        Node * _next;
    };
    Node* _last;
    int _size;
public:
    List(int =0);
    friend ostream& operator<<(ostream&, const
                                    List&);
    int size() const { return _size; }
    bool empty() const { return _size==0; }
    char last () const {return _last->_ch; }
    char kill(int);
};

int main()
{ int n, skip;
    cout<<"Nhập vào số lính bị bao vây:"; cin>>n;
    List list(n);
    cout<<"\t " <<list;
    cout<<"Nhập vào số bo qua:"; cin>>skip;
```

```

        while(list.size()>1)
        {
cout<<"\t"<<list.kill(skip)<<"killed "<<list;
        }
cout<<list.last()<<"Nhung nguoi con song
sot:\n";
        }

List::List(int n):_size(n)
{ _last=new Node('A'+n-1);
Node *p=_last;
for(int i=0; i<n-1; i++)
p=p->_next=new Node('A'+i);
p->_next=_last;
}
ostream& operator<<( ostream& ostr, const List&
list)
{ ostr<<"\t size="<<list._size;
if(list.empty()) cout<<":\t()\n";
else
{ List::Node* p=list._last->_next;
ostr<<": (" <<p->_ch;
for(p=p->_next; p!=list._last->_next; p=p->_next)
ostr<<","<<p->_ch;
ostr<<") \n";
}
return ostr;
}
char List::kill(int k);
{ static Node* p=_last;
for(int i=0; i<k; i++)
-
-
```

```

    p=p->_next;
    Node* temp=p->_next;
    char killed=temp->_ch;
    p->_next=temp->_next;
    if(temp==_last) _last=p;
    delete temp;
    --_size;
    return killed;
}

```

*Kết quả chạy chương trình:*

Nhap vao so linh bi bao vay: 10

size=10 : (A, B, C, D, E, F, G, H, I, J)

Nhap vao so bo qua: 3

D kilied size=9 : (A, B, C, E, F, G, H, I, J)

H kilied size=8 : (A, B, C, D, E, F, G, I, J)

B kilied size=7 : (A, C, D, E, F, G, H, I, J)

G kilied size=6 : (A, C, E, F, I, J)

C kilied size=5 : (A, E, F, I, J)

J kilied size=4 : (A, E, F, I)

I kilied size=3 : (A, E, F)

A kilied size=2 : (E, F)

F kilied size=1 : (E);

Nhung nguoi con song sot: E

4.17. Mô phỏng một hàng đợi ưu tiên để in tài liệu dùng chung một máy in trong mạng LAN

Các máy in riêng lẻ trong mạng LAN được sử dụng để phục vụ cho các yêu cầu in của tất cả các máy tính trên mạng. Nếu thuật toán “vào trước được phục vụ trước” được sử dụng, thì máy in có thể sử dụng một hàng đợi FIFO bình thường để giữ tài liệu in được đưa đến trong lúc nó đang bận in. Một thuật toán phổ biến khác sẽ xử lý các tài liệu in ngắn hơn, các tài liệu in dài hơn thì

phải chờ in trong hàng đợi in. Điều này đòi hỏi phải có *hàng đợi ưu tiên*, ở đó các tài liệu ngắn hơn được ưu tiên hơn so với các tài liệu dài hơn.

Chương trình sau đây mô phỏng tiến trình này.

```
# include <iostream.h>
# include <queue.h> // priority_queue class template tiền xử lý
# include "Random.h" // Random class tiền xử lý
using namespace std;

const int TICKS=60; // 60 giây
const float MTBA=4; // thời gian trung bình giữa 2 lần tài liệu đến
const float MNOP=5; // số trung bình các trang cho 1 lần in
class Exp: public Random // phát sinh các số ngẫu nhiên phân
    bố mõ (giống bài 4.12)
{
public:
Exp(float mean): _mean(mean) {}
float next() { return -_mean*log(1.0-real()); }
private:
float _mean;
};

class Job // biểu thị những tài liệu cần in
{
friend bool operator< (const Jobs& job1, const
Job& job2) // các tài liệu nhỏ có độ ưu tiên cao
{ return (job1._pages > job2._pages); }
friend ostream& operator<< (ostream& ostr, const
Job& job)
{ ostr << "Job #" << job._id << "(" << job._pages
<< "pp)";
return ostr;
}
```

```
static int _n;           // đánh số thứ tự các tài liệu in: #0, #1, #2, ...
static Exp _gen;         // đếm số trang in
public:
Job(int time=0): _id(_n++), _arrived(time),
    _pages(1+int(_gen.next())) {}  

int id() { return _id; }
void process () {--_pages;} // mô phỏng việc in 1 trang
bool is_finished() { return (_pages ==0); }
protected:
int _id;                 // mã số trang
int _arrived;            // thời gian đến của tài liệu in
int _pages;               // số lượng trang in
};
int Job:: _n = 0;          // khởi trị biến đếm trang in
Exp Job::_gen(MNOP);     // định nghĩa bộ sinh biến đếm trang
class Source              // lớp các đối tượng của công việc in
{
public:
Source(float iat) : _gen(Exp(iat)),
    _arrival_cime(0) {}
Job* arrival(int t)// trả tới tài liệu mới nếu nó tới vào thời điểm t
{
    Job* job = 0; if (t == _arrival_time)
    { job = new Job(t);
        _arrival_time += int(_gen.next())+1;
        // cộng thêm một số ngẫu nhiên
    }
    return job;
}
protected:
```

```

    Exp _gen;           // bộ sinh số ngẫu nhiên
    int _arrival_time; // thời gian đến của tài liệu kế tiếp
};

string size(priority_queue<Job>); // trả về 1 xâu biểu thị
// kích thước của hàng đợi

int main()
{
    Source source(MTBA)      // các tài liệu in phát sinh với thời
    // gian đến trung bình là MTBA
    priority_queue<Job> pq; // priority queue của công việc in
    Job* current_job=0;     // địa chỉ 0 biểu thị không có tài
    // liệu in hiện thời

    int cjid;               // chỉ số biểu thị tài liệu in hiện hành
    for (int t=0; t<TICKS; t++) // mỗi lần lặp biểu thị một tích
    // tắc của đồng hồ (1 giây)

    {
        cout << "\n" << t <<"; // xuất thời gian hiện hành
        Job* new_job = source.arrival(t); // phát sinh tài
        // liệu mới cần in

        if (new_job != 0)
        {
            pq.push(*new_job); // đẩy nó vào hàng đợi
            cout << *new_job << " arrived: ";
            // và thông báo về nó
        }

        cout << size(pq); // thông báo kích thước của hàng đợi
        if (current_job == 0 && !pq.empty()) // bắt đầu 1 tài
        // liệu in tiếp từ hàng đợi
        {
            Job job = pq.top(); // lưu cát độ ưu tiên cao vào hàng đợi
            pq.pop();           // hủy tài liệu đã in khỏi hàng đợi
            current_job = &job; // rồi lưu địa chỉ
            cjid = current_job->id(); // và chỉ số của tài liệu
        }
    }
}

```

```

cout << ", start #" << cjid << " : " <<
size(pq);           // thông báo
}

if (current_job != 0)           // có một tài liệu hiện hành
{ current_job->process(); // xử lý tài liệu ấy
  cout << ", process #" << cjid; // thông báo
  if (current_job->is_finishedf())
    { current_job = 0; // xóa tài liệu hiện hành
      cout << ", end #" << cjid; // thông báo kết quả
    }
}
}

string size(priority_queue<Job> pq)
{ string s=" [";
  for (int i=0; i<pq.size(); i++)
  s += "*'";
  s += " ]";
  return s;
}

```

*Phân tích code:*

Lớp Exp ở đây tương tự như lớp được sử dụng trong bài tập 4.12.

Lớp Job tạo ra các đối tượng biểu thị cho các tài liệu in được gửi đến máy in trên mạng LAN. Toán tử so sánh `<` phải có đối với các đối tượng được lưu trong một priority\_queue. Lưu ý rằng job1 có mức độ ưu tiên thấp hơn so với job2 nếu nó có nhiều trang hơn. Toán tử xuất `<<` được sử dụng để cho biết thông tin về các tài liệu in khi chúng được đưa vào phần mô phỏng. Mỗi tài liệu in đều có một số nhận dạng riêng `_id`, thời gian đến `_arrived`, và kích cỡ hiện tại `_pages`. Số nhận dạng của nó `_id` được cung cấp bởi trường static `_n`, và kích cỡ ban đầu của nó `_pages` được tạo bởi đối tượng static `_gen`. Đối tượng đó tạo ra các số ngẫu nhiên được phân bố ngày càng tăng với giá trị trung bình là

MNOP, được bắt đầu ở 5 trang. Lưu ý rằng 10 tài liệu in đầu tiên có 7, 3, 4, 3, 2, 2, 7, 11, 1, và 6 trang.

Trường `_pages` được bắt đầu tại `1+int (_gen, .next())` để bảo đảm rằng không có tài liệu nào bắt đầu với 0 trang. Hàm `process()` mô phỏng việc in một trang, và hàm `is_finished()` xác định xem tất cả các trang đã được in hay chưa.

Lưu ý rằng hai trường `Job: :_n` và `Job: :_gen` phải được xác định và được bắt đầu ở bên ngoài lớp `Job`. Số nguyên `_n` được khởi tạo bằng 0 đối với phần nhận dạng tài liệu in đầu tiên, và đối tượng `_gen` được khởi tạo bởi bộ tạo `Exp`, sau khi chuyển đến nó hằng MNOP 5 trang.

Lớp `Source` được sử dụng để cụ thể hóa lớp đơn tạo ra tất cả các tài liệu in. Nó sử dụng trường `_gen` để tạo các số ngẫu nhiên được phân bố ngày càng tăng đối với số lần giữa các tài liệu in được gửi đến, với giá trị trung bình MTBA 4 giây giữa những lần gửi đến. Hãy lưu ý rằng 10 tài liệu in đầu tiên được gửi đến với thời gian 0, 6, 8, 11, 13, 15, 17, 23, 32 và 33 giây: vì vậy khoảng thời gian được tạo là 6, 2, 3, 2, 2, 2, 6, 9, và 1 giây, trung bình là  $33/9 = 3,67$  giây giữa các lần gửi đến. Biểu thức `int (_gen, .next()) + 1` được thêm vào mỗi thời gian gửi đến để bảo đảm rằng không có hai tài liệu in được cùng một lúc. Lưu ý rằng bộ tạo `Source` sử dụng biểu thức `Exp (iat)` để khởi tạo đối tượng `_gen` của nó. Biểu thức này viễn dẫn bộ tạo `Exp`, sau khi gửi đến nó thời gian `iat` của các lần gửi đến. Trong hàm `main()`, giá trị này là hằng MTBA 4.

Hàm `Source : : arrival()` được sử dụng để tạo ra các tài liệu in. Mỗi lần nó được gọi, nó sẽ sử dụng biểu thức (`t == _arrival_time`) để xác định xem đó có phải là lúc tạo ra một tài liệu in mới không. Nếu đúng thì biểu thức `new Job(t)` được dùng để tạo một tài liệu in mới, sau khi xác định nó với thời gian hiện tại là `t`. Tiếp theo nó cập nhật trường `_arrival_time`, sau khi xác lập nó bởi thời gian của tài liệu in tiếp theo được gửi đến. Hàm này cho ra một pointer `Job*`, mà nó trả đến một tài liệu in mới hay 0 (tức là rỗng), tùy theo đó có phải là thời gian để một tài liệu in được tạo không. Hàm `main()` gọi hàm `Source: :arrival()` một lần đối với mỗi lần lặp trong vòng lặp `for` của nó. Lưu ý rằng hàm `arrival()` cho ra 0 ngoại trừ ở các lần lặp 0, 6, 8, 11, 13, 15, 17, 23, 32, 33,...

Chương trình `main` dùng một vòng lặp `for` để thực hiện việc mô phỏng. Nó lặp lại một lần đối với mỗi “tích tắc” đồng hồ. Trước tiên nó gọi `Source: :arrival()` để nhận tài liệu in tiếp theo nếu nó được gửi đến và đẩy nó lên hàng đợi ưu tiên `pg`. Sau đó nó kiểm tra xem có một tài liệu in đang được in không, nếu không và nếu hàng đợi không có tài liệu nào đang chờ in thì nó sẽ xóa tài

liệu in có mức độ ưu tiên cao nhất (tài liệu có số trang ít nhất) từ hàng đợi và làm cho nó trở thành tài liệu in hiện hành bằng cách gán địa chỉ của nó vào pointer current\_job. Cuối cùng, nếu có một tài liệu in hiện hành thì nó gọi hàm process() để mô phỏng việc in một trang của tài liệu đó và sau đó xóa nó, nếu nó đã được thực hiện xong.

#### 4.22. Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#define true 1
#define false 0
//-----
struct node
{
    int info;
    node* next;
};
//-----
//Khoi tao danh sach, dat phead=NULL
void init(node* &phead)
{
    phead=NULL;
}
//-----
void clear(node* &phead)      //Giải phóng tất cả các nút
                                //trong danh sách liên kết đơn
{
    node *p, *p1;
    p=phead;
    while (p!=NULL)
        {p1=p;
```

```

    p=p->next;
    delete p1;
}
phead=NULL;
}

//-----
int empty(node* &phead)           //Trước khi chèn, tìm, hoặc
{ return(phead==NULL);           //xóa cần kiểm tra
}                                     //xem danh sách liên kết
                                       //đơn có rỗng không?
}

//-----
node* search(node* &phead, int k)
{ node* p=phead;
  while(p!=NULL)
  { if(p->info==k) return(p);
    p=p->next;
  }
  return(NULL);
}

//-----
void insert(node* &phead, int k)   //Chèn phần tử có khóa
{ node *pp,*p,*p1;                 //k vào danh sách
  p=search(phead,k);
  if(p!=NULL) {printf("\nKhoa %d da co, khong chen
                      duoc!",k);return;}
  pp = new node;
  pp->info=k;pp->next=NULL;
  if(phead==NULL) {phead=pp;return;}
}

```

```

p1=NULL;
p = phead;
while(p!=NULL)
{
    p1=p;
    p=p->next;
}
//Sau vòng lặp thì p1 là nút cuối cùng
p1->next = pp;
};

//-----
void dele(node* &phead, int k)      //Xóa nút có khóa k do
                                         phead trỏ đến
{
    node *p,*p1,*q;
    if(phead==NULL) return;
    if(phead->info==k)
    {p=phead;phead=phead->next;delete p;return;}
    p1=phead;
    p=phead->next;
    while(p!=NULL)
    {
        if(p->info==k)
        {q=p; p1->next=p->next; delete q; return;}
        p1=p;
        p=p->next;
    }
    printf("\nKhoa %d khong co, khong xoa duoc!",k);
};

//-----
void SapXepDc(node *phead)          //Sắp xếp bằng phương pháp
                                         Đổi chỗ trực tiếp
{
    node *p,*p1,*p2;int t;
}

```

```

if(empty(phead)) return;
p1=phead;
//pt truoc p,plt truoc p1
while(p1!=NULL)
{ p=p1;
  p2=p1->next;
  while(p2!=NULL)
  { if(p2->info<p1->info)
    { t=p1->info;p1->info=p2->info;p2->info=t; }
    p2=p2->next;
  }
  p1=p1->next;
}
}

//-----
void traverse(node* &phead)           //Duyệt danh sách liên
//kết do phead trỏ đến
{ node* p=phead;
  printf("\nCac phan tu trong danh sach:\n");
  while(p!=NULL)
  { printf("%4d",p->info);
    p=p->next;
  }
}

//-----
void Input(node* &phead)           //Nhập dữ liệu bằng tay hoặc tự
//động ngẫu nhiên
{ clrscr();int m,i,int k;
  printf("\nNhap du lieu vao danh sach:");
  printf("\n1. Nhap truc tiep");
}

```

```
printf("\n2. Nhập từ đồng ngẫu nhiên");
printf("\n\n Hãy chọn 1 hoặc 2: ");
fflush(stdin);
char ch=getch();
printf("\nChọn biết số phần tử cần đưa vào danh
sách: ");
fflush(stdin);
scanf("%d", &m);
if(ch=='1')
    {printf("\nHãy nhập %d số: ",m);
     for(i=0;i<m;i++)
        { scanf("%d", &k);
         insert(phead,k);
        }
    }
else
    {randomize();
     for(i=0;i<m;i++)
        { k=random(10*m);
         insert(phead,k);
        }
    }
};

//=====
void main()
{ clrscr();int k;node* phead, *p;
while(true)
{clrscr();
 printf("\n 1. Khởi tạo danh sách");
```

```
printf("\n 2. Chen (nhap) cac phan tu vao
      danh sach.");
printf("\n 3. Xoa 1 phan tu trong danh sach");
printf("\n 4. Tim 1 phan tu trong danh sach");
printf("\n 5. Sap xep dsdk don bang t_toan
      Doi cho Truc tiep.");
printf("\n 6. Duyet danh sach");
printf("\n 0. Thoat (khong lam gi nua!)");
printf("\n Hay nhan phim tu 1 -> z de chon: ");
char chon=toupper(getch());
if(chon=='0') break;
switch(chon)
{
    case '1': init(phead);break;
    case '2': Input(phead);break;
    case '3': printf("\n Nhap khoa can xoa: ");
                fflush(stdin);scanf("%d",&k);
                dele(phead,k); break;
    case '4': printf("\n Nhap khoa can tim:
                ");scanf("%d",&k);
    p=search(phead,k);
    if(p!=NULL)
        printf("\n Phan tu tim thay co khoa
               la:%d",p->info);
    else printf("\n Khong tim thay khoa %d trong
               ds",k);break;
    case '5': SapXepDc(phead);
                traverse(phead);break;
    case '6': if(empty(phead)) printf("\n Danh
               sach rong");
    else traverse(phead);break;
}
```

```
    printf("\n\nNhan phim bat ky de tiep tuc");
    getch();
}
}
```

#### 4.23. Code:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "string.h"
#include <iostream.h>
#include <dos.h>
#define true 1
#define false 0
struct ts
{ char hten[20]; char msv[8]; char tennganh[40];
  float t,l,h,sum;
  ts *pNext;
};
int sosv; ts *pHead, *pcurrent, *pTail;
void khoitao();
void AddTail();
void nhap();
void xemdo();
void SapXepDc();
void xemtoands();
void Tim();
//=====
void main()
{const int F1=59,F2=60,F3=61,F4=62,F5=63,F6=64,F10=68;
 ↑
//Những độc giả lần đầu học môn CTDL&TT hãy chú ý dòng này!
```

```
char chon;
while(true)
{ clrscr();
  printf("\nF1. Khoi tao.");
  printf("\nF2. Nhap.");
  printf("\nF3. Xem cac sv (do) co tong
         diem>=18.");
  printf("\nF4. Sap xep tong diem tang.");
  printf("\nF5. Xem toan ds.");
  printf("\nF6. Tim theo ma so sv.");
  rintf("\nF10. Ket thuc");
  printf("\n Nhan phim F1 - F10 de chon");
  printf("\n");
  chon=getch();
  if(chon==0) chon=getch();
  if(chon==F10) break;
  switch(chon)
  { case F1: Khoitao();break;
    case F2: Nhaph();break;
    case F3: Xemdo();break;
    case F4: SapXepDc(); break;
    case F5: Xemtoands(); break;
    case F6: Tim(); break;
  }
  printf("\n");
  printf("Nhan phim bat ky de tiep tuc");
  getch();
}
}

//=====
```

```
void khoitao()
{
    pHead=NULL;
    soSV=0; pcurrent=NULL;
}

//=====
int empty(ts * &pHead)
{
    return(pHead==NULL);
}

//=====
void AddTail()
{
    ts *p;
    if(pHead==NULL)
    {
        pHead=p;
        pTail=pHead;
    }
    else
    {
        pTail->pNext=p;
        pTail=p;
    }
}

//=====
void nhap()
{
    int i,sl;
    ts *p1,*p, *tmp;
    p=p1=pHead;
    while(p!=NULL)
    {
        p1=p;
        p=p->pNext;
    }
    cout<<endl<<"Nhập bao nhiêu người?"; cin>>sl;
```

```

for(i=1;i<=sl;i++)
{
    tmp=new ts;tmp->pNext=NULL;
    cout<<endl<<"Nhập sinh viên thứ "<<i<<":" ;
    cout<<endl<<"Họ tên: ";gets(tmp->hten);
    cout<<endl<<"Mã số: "; gets(tmp->msv);
    cout<<endl<<"Tên ngành: "; gets(tmp->tennganh);
    cout<<endl<<"Nhập điểm toán: "; cin>>tmp->t;
    cout<<endl<<"Nhập điểm lý: "; cin>>tmp->l;
    cout<<endl<<"Nhập điểm hóa: "; cin>>tmp->h;
    tmp->sum=tmp->t+tmp->l+tmp->h;
    if(p1==NULL)
        {p1=pHead=tmp;}
    else
        {p1->pNext=tmp;p1=tmp;}
}
sosvt+=sl;
pcurrent=p1;
AddTail();
}

//=====
void xemdo()
{
    int i; ts *p=pHead;
    printf("Danh sách thi sinh do:\n\n");
    printf(" Ten   Mav  Dtoan  Dly  Dhoa  Dtong:
          \n");
    while(p!=NULL)
        { if (p->sum>=18)
            {cout<<p->hten<<"\t"<<p->msv<<"\t"<<p-
                >t<<"\t"<<p->l<<"\t"<<p->h
                <<"\t"<<p->sum;

```

```
    printf("\n");
}
p=p->pNext;
}
}

//-----
void SapXepDc()          //Sắp xếp theo thuật toán đổi chỗ trực tiếp
{ts *p,*p1,*p2;float t;
 if(empty(pHead)) return;
 p1=pHead;
 //pt truoc p,plt truoc p1
 while(p1!=NULL)
 {p=p1;
 p2=p1->pNext;
 while(p2!=NULL)
 { if(p2->sum<p1->sum)
 { t=p1->sum;p1->sum=p2->sum;p2->sum=t; }
 p2=p2->pNext;
 }
 p1=p1->pNext;
 }
}

void xemtoands()
{ int i; ts *p=pHead;
 printf("Toan Ds thi sinh xep theo Tong diem tang
 dan:\n\n");
 printf(" Ten   Msv  Dtoan   Dly   Dhoa   Dtong:\n");
 while(p!=NULL)
 {cout<<p->hten<<"\t"<<p->msv<<"\t"<<p-
 >t<<"\t"<<p->l<<"\t"<<p->h
```

```
        <<"\t"<<p->sum;

        printf("\n");
        p=p->pNext;
    }

void Tim()
{
    ts *p; char x[8];
    printf("Cho biet ma sv can tim:"); scanf("%s", &x);
    p=pHead;
    while(p!=NULL&&strcmp(p->msv, x)!=0)
        p=p->pNext;
    if (p==NULL)
        printf("Khong tim thay Sv co ma x trong
               ds.", p->msv);
    else
        printf("Da thay Sv co ma %s trong ds.", p-
               >msv);
}
```

## Chương 5

# CẤU TRÚC CÂY (TREE STRUCTURE)

### 5.1. CÁC KHÁI NIỆM CƠ BẢN VỀ CÂY

Cây là một cấu trúc phân cấp của một tập các đối tượng nào đó. Ví dụ quen thuộc là cây thư mục trong các kho (store) lưu trữ dữ liệu trên máy tính. Từ năm 1857, cây đã được nhà toán học Anh là Arthur Cayley dùng để định nghĩa và biểu diễn các dạng hợp chất khác nhau của hóa học; từ đó cây được dùng trong nhiều lĩnh vực khoa học khác nhau, trong đó có Tin học.

Có một vài cách định nghĩa cây, dưới đây là cách định nghĩa đệ quy về cây.

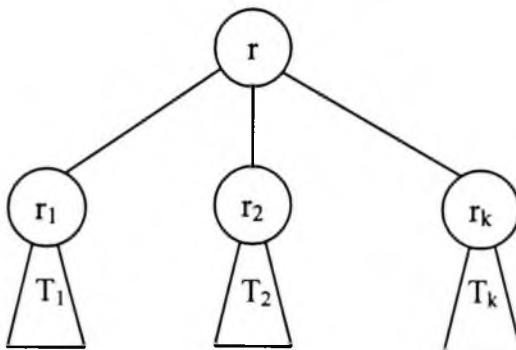
#### 5.1.1. Định nghĩa đệ quy về cây

Tập hợp có một phần tử gọi là cây (Phần tử của cây còn gọi là nút (node) hoặc đỉnh). Nút *đuy nhất* này của cây gọi là *gốc* (Root) của cây.

Ta gọi các tập  $T_1, T_2, \dots, T_k$  ( $k \geq 1$ ) là các cây với các gốc tương ứng là  $r_1, r_2, \dots, r_k$ . Giả sử các cây  $T_i$  rời nhau ( $T_i \cap T_j = \emptyset, i \neq j$ ). Cho  $r$  là một nút không thuộc các cây  $T_i$  ( $i=1, 2, 3, \dots, k$ ). Khi đó tập  $T$  chứa nút  $r$  và tất cả các nút của các cây  $T_i$  gọi là cây  $T$  với gốc  $r$ .

Các cây  $T_i$  gọi là *cây con* của gốc  $r$ .

Các gốc  $r_i$  của các cây con  $T_i$  đều có cung nối với gốc  $r$  (xem hình 5.1).



Hình 5.1. Cây  $T$  với gốc  $r$  và các cây con của các gốc tương ứng  $r_1, r_2, \dots, r_k$

Sau đây là cách định nghĩa phi đệ quy rất ngắn gọn về cây.

### 5.1.2. Định nghĩa phi đệ quy về cây

Một đồ thị vô hướng *liên thông không có chu trình* (đơn) gọi là *cây*.

### 5.1.3. Cây và những khái niệm liên quan

Từ cấu trúc của cây và định nghĩa của nó ta thấy *mỗi nút* của cây là *gốc* các cây con của nó.

- *Bậc của một nút* là số cây con của nút ấy. Nút có *bậc bằng không* gọi là *nút lá*.

- Nếu nút a có một cây con với gốc là b thì ta nói *a là cha của b và b là con của a*. (xem hình 5.2). Như vậy dựa vào khái niệm “nút cha, nút con” ta có thể nói: *bậc của một nút là số các nút con của nó*, còn *lá* là *nút không có con*. Các nút có ít nhất một con gọi là *các nút trong*. Các nút của cây hoặc là lá hoặc là *nút trong*.

- Các nút có cùng một cha gọi là *các nút anh, em*.

- Dãy các nút  $a_1, a_2, \dots, a_n$  ( $n \geq 1$ ) mà  $a_i$  ( $i=1, 2, \dots, n-1$ ) là cha của  $a_{i+1}$  được gọi là *đường đi từ  $a_1$  đến  $a_n$* . Luôn tồn tại một đường đi duy nhất từ gốc đến một nút bất kỳ trong cây.

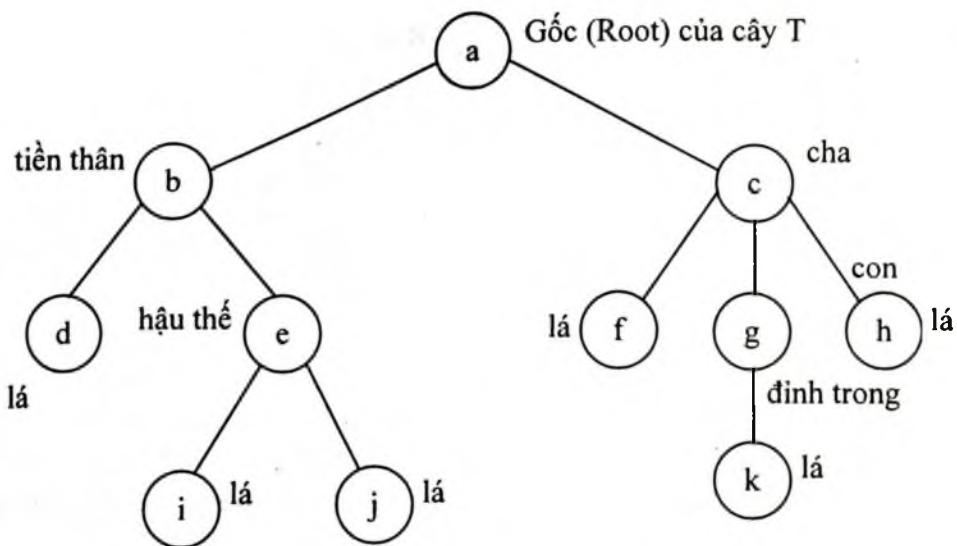
- Nếu có một đường đi từ nút a đến nút b với độ dài  $k \geq 1$  thì ta gọi a là *tiền thân* của b và b là *hậu thế* của a.

- Từ định nghĩa và cấu trúc của cây ta thấy mỗi nút a bất kỳ của cây T là gốc của một cây nào đó, ta gọi cây này là cây con của T. Cây con này có gốc là a và gồm tất cả các nút là *hậu thế* của a. Ví dụ một cây con của cây T ở hình 5.2 là cây  $T_1$  gồm gốc của nó là b và các nút  $\{d, e, i, j\}$ .

- *Độ cao h của một nút* thuộc cây là độ dài lớn nhất của đường đi từ nó đến một lá. Ví dụ ở hình 5.2 ta có độ cao h của b là  $h(b)=2$ . *Độ cao H của gốc là độ cao của cây*.

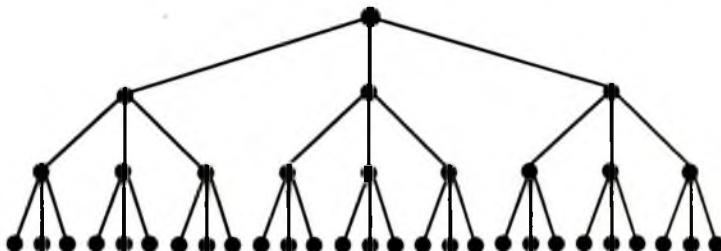
- *Mức* của đỉnh bất kỳ là *độ dài đường đi từ gốc đến nút đó*. Chẳng hạn mức của nút e ở hình 5.2 là 2 (*không ảnh hưởng tới tính tổng quát của vấn đề đang xét ta giả sử độ dài đường đi trực tiếp giữa hai nút kề liền nhau có độ dài là 1 đơn vị dài*).

- Như vậy mức của gốc của cây T trong hình 5.2 bằng 0.



Hình 5.2. Kiến trúc cây T và các thành phần của nó

- **Cây hoàn chỉnh:** Một cây được gọi là **hoàn chỉnh** nếu tất cả các nút trong của nó có cùng một bậc và mọi lá của nó có cùng mức.

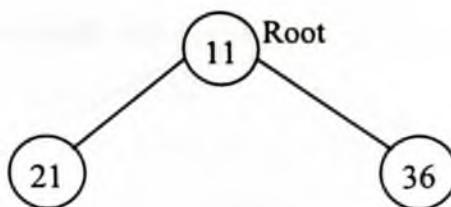


Hình 5.3. Cây hoàn chỉnh

- **Cây quyết định:** Cây có gốc mà **mỗi đỉnh trong** của nó ứng với **một quyết định** và **mỗi cây con** của các đỉnh này ứng với **một kết cục** của quyết định gọi là cây quyết định. (Xem minh họa về khái niệm trên đây tại đáp án các bài tập 5.9 và 5.11 ở cuối chương này).

#### 5.1.4. Cây được sắp

Vì cây cũng là một cấu trúc dữ liệu nên các nút của nó cũng dùng để lưu trữ dữ liệu. Nếu dữ liệu ở các nút của một cây được sắp theo một thứ tự nào đó thì cây ấy gọi là cây được sắp. Hình 5.4 minh họa cho một phương án cây được sắp. Từ đây về sau ta chỉ quan tâm đến các cây được sắp, do vậy sẽ nói là cây.

*Hình 5.4. Cây được sắp*

### 5.1.5. Nút con trưởng và nút anh liền kề

Giả sử cây T có nút a với các nút con  $b_i$  ( $i=1, 2, 3, \dots, k$ ) của a được sắp theo thứ tự  $b_1, b_2, \dots, b_k$  ( $k \geq 1$ ). Khi đó ta nói  $b_1$  là con trưởng của a, còn  $b_i$  là anh liền kề của  $b_{i+1}$  ( $b_{i+1}$  là em liền kề của  $b_i$ ). ta còn nói với  $i < j$  thì  $b_i$  ở bên trái  $b_j$  ( $b_j$  ở bên phải  $b_i$ ). Như vậy nếu  $b_i$  ở bên trái  $b_j$  thì tất cả hậu thế của  $b_i$  cũng ở bên trái mọi hậu thế của  $b_j$ .

### 5.1.6. Cây gán nhãn

Cây gán nhãn là một phép tương ứng 1-1 giữa tập các dữ liệu (số, chữ cái, xâu, bản ghi,...) và các nút của cây, các dữ liệu đó gọi là nhãn (Label).

### 5.1.7. Rừng

Rừng F là một tập hợp các cây  $F = \{T_1, T_2, \dots, T_n\}$

### 5.1.8. Định lý 1

*Một đồ thị vô hướng là một cây nếu giữa mọi cặp đỉnh của nó luôn tồn tại đường đi đơn duy nhất.*

Để chứng minh định lý này ta cần dùng hai bỗ đề đã được tìm hiểu trong môn “Lý thuyết đồ thị” (Độc giả có thể tìm hiểu vấn đề này ở tài liệu tham khảo [8])

a. **Bỗ đề 1.** Giữa mọi cặp đỉnh phân biệt của một đồ thị vô hướng liên thông luôn tồn tại đường đi đơn.

b. **Bỗ đề 2.** Gọi  $P_1$  và  $P_2$  là hai đường đi đơn giữa các đỉnh  $u$  và  $v$  trong đơn đồ thị  $G$  không chứa cùng một tập các cạnh. Chứng tỏ rằng khi đó tồn tại một chu trình đơn trong  $G$ .

### c. Chứng minh định lý 1

Giả sử  $T$  là một cây, khi đó  $T$  là một đồ thị liên thông không có chu trình. Gọi  $x, y$  là 2 đỉnh của  $T$ . Vì  $T$  liên thông nên theo bỗ đề 1 có đường đi đơn giữa 2 đỉnh này. Đường đi này là *duy nhất* vì nếu có đường đi thứ hai từ  $x$  đến  $y$  thì đường đi phát sinh từ hợp của đường đi thứ nhất từ  $x$  tới  $y$  với đường đi thứ hai

từ y đến x nhận được bằng cách đảo ngược đường đi thứ hai từ x đến y sẽ tạo thành chu trình. Từ đó theo bô đề 2 suy ra có chu trình đơn trong T. Vì vậy giữa mọi cặp đỉnh của cây luôn có đường đi đơn duy nhất.

Đảo lại, giả sử giữa hai đỉnh bất kỳ của đồ thị T luôn có đường đi đơn duy nhất. Khi đó T là liên thông, hơn nữa T không thể có chu trình đơn. Giả sử ngược lại có chu trình đơn chứa hai đỉnh x, y.

Khi đó rõ ràng có hai đường đi giữa x và y. Vì đường đi thứ nhất là phần từ x đến y thuộc chu trình, phần thứ hai là đoạn còn lại của chu trình song theo hướng ngược lại. Nghĩa là giữa x và y có hai đường đi đơn. Điều này không thể được. Vì vậy T là đồ thị liên thông không có chu trình đơn (tức T là cây).

## 5.2. ĐỊNH NGHĨA CÂY M\_PHÂN

Ta gọi r là gốc của cây T.

- Cây T được gọi là cây m\_phân nếu tất cả các đỉnh trong của nó *không có nhiều hơn m con*.

- Cây T được gọi là cây m\_phân *đầy đủ* nếu *mọi* đỉnh trong của nó *có đúng m con*

- Với m=2 cây được gọi là cây nhị\_phân.

## 5.3. CÁC TÍNH CHẤT CỦA CÂY M\_PHÂN

### 5.3.1. Định lý 2

*Cây có n nút có đúng n-1 cạnh.*

**Chứng minh.** Chọn nút r làm gốc của cây, ta sẽ lập một phép tương ứng 1-1 giữa các cạnh với các nút khác r bằng cách gán nút cuối của một cạnh với chính cạnh này. Vì có n-1 đỉnh khác r nên cây có đúng n-1 cạnh.

### 5.3.2. Định lý 3

*Cây m\_phân đầy đủ với i nút trong sẽ có cả thảy  $n=m \cdot i + 1$  nút.*

**Chứng minh.** Mỗi nút trừ gốc là con của một nút trong. Vì mỗi một trong i nút trong có m con (giả thiết) nên có  $m \cdot i$  nút khác gốc. Đến đây ta được cây có tất cả  $m \cdot i + 1$  nút (kể cả gốc).

### 5.3.3. Định lý 4

*Cây m\_phân đầy đủ với:*

$$\text{a. } n \text{ nút có } k = \frac{n-1}{m} \text{ nút trong và } l = \frac{(m-1)n+1}{m} \text{ lá}$$

b. i nút trong có  $n=m.i+1$  nút và  $l=(m-1)i+1$  lá

c. Có  $n=\frac{ml-1}{m-1}$  nút và  $i=\frac{l-1}{m-1}$  nút trong

**Chứng minh.** Gọi  $n$  là số nút,  $i$  là số nút trong,  $l$  là số nút lá. Ta có thể chứng minh 3 phần của định lý này nhờ dùng đẳng thức  $n=m.i+1$  của định lý 3 cùng với đẳng thức  $n=l+i$ . Sở sỉ làm được việc này vì hiển nhiên với mỗi nút tùy ý của cây thì nó hoặc là nút trong hoặc là lá. Chứng minh mục a:

Từ đẳng thức của định lý 3 ta có:

$$i = \frac{n-1}{m} \quad (\text{xong ý thứ nhất!})$$

Thay  $i$  tìm được như trên vào hệ thức  $n=l+i$

$$\text{ta được } l=n-i=n-\frac{n-1}{m}=\frac{(m-1)n+1}{m} \quad (\text{xong ý thứ hai})$$

Hai phần còn lại độc giả tự chứng minh!

#### 5.3.4. Định lý 5

Có nhiều nhất  $m^h$  lá trong cây m\_phân với độ cao h.

**Chứng minh.** Ta sẽ chứng minh bằng quy nạp theo chiều cao. Đầu tiên xét cây có chiều cao  $h=1$ , những cây như thế ngoài gốc có không hơn  $m$  con và mỗi con là một lá. Nghĩa là có không quá  $m^1 = m$  lá trong cây có chiều cao  $h=1$ .

Giả thiết quy nạp là: giả sử kết quả đúng với cây m\_phân có chiều cao nhỏ hơn  $h$ . Gọi  $h$  là chiều cao của cây m\_phân T, các lá của T là các lá của các cây con nhận được từ T bằng cách xóa các cạnh nối từ gốc tới các nút ở mức 1.

Mỗi một cây trong các cây con này có chiều cao không quá  $h-1$ . Như vậy theo giả thiết quy nạp mỗi cây con có nhiều nhất  $m^{h-1}$  lá. Vì có nhiều nhất  $m$  cây con như vậy mà mỗi cây có nhiều nhất  $m^{h-1}$  lá suy ra có nhiều nhất  $m.m^{h-1} = m^h$  lá trong cây T.

#### 5.3.5. Hệ quả

Nếu cây m\_phân với chiều cao  $h$  có  $l$  lá thì  $h \geq \lceil \log_m l \rceil$ . Nếu cây m\_phân đầy đủ và cân đối thì  $h=\lceil \log_m l \rceil$  (ký hiệu  $\lceil x \rceil$  là số nguyên nhỏ nhất lớn hơn hay bằng  $x$ ).

**Chứng minh.** Từ định lý 5 ta có  $l \leq m^h$ . Lấy logarit hai về theo cơ số  $m$  ta có  $\log_m l \leq h$ . Vì  $h$  là số nguyên dương nên  $h \geq \lceil \log_m l \rceil$ . Vậy giờ giả sử cây là

cân đối (*cây cân đối là cây mà mọi lá đều ở cùng một mức*). Khi đó mọi lá đều ở mức  $h$  hoặc  $h-1$ . Vì chiều cao của chúng là  $h$  nên có ít nhất một lá ở mức  $h$ .

Từ đó suy ra phải có hơn  $m^{h-1}$  lá (**Bổ đề: Cây m\_phân đầy đủ cân đối với chiều cao h có nhiều hơn  $m^{h-1}$  lá**). Vì  $l \leq m^h$  ta có  $m^{h-1} < l \leq m^h$ . Lấy logarit hai về theo cơ số  $m$  ta được  $h-1 < \log_m l \leq h$ , tức là:

$$h = [\log_m l].$$

## 5.4. CÂY NHỊ PHÂN - CNP

### 5.4.1. Định nghĩa phi đệ quy về cây nhị phân (Binary tree - BT)

Như mục 5.2 đã trình bày tổng quát về định nghĩa cây m\_phân, với  $m=2$  ta có cây 2\_phân (cây nhị phân) Tuy nhiên để rõ hơn về đặc điểm của cây này ta nêu một cách định nghĩa khác về cây 2\_phân:

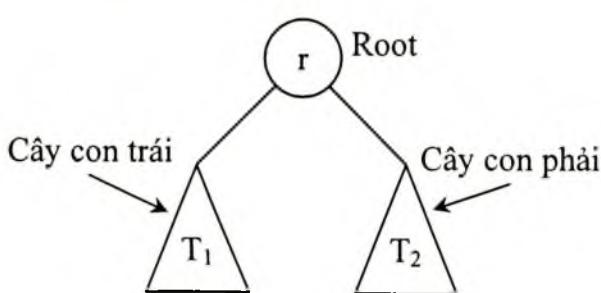
*Cây nhị phân là cây gồm các nút kết nối với nhau theo quan hệ “cha-con” và mỗi nút cha chỉ có tối đa 2 con.*

### 5.4.2. Định nghĩa đệ quy về cây nhị phân

Cây nhị phân là tập hợp hữu hạn các nút được định nghĩa đệ quy như sau:

- Một tập rỗng là cây nhị phân
- Cho hai tập  $T_1, T_2$  rời nhau ( $T_1 \cap T_2 = \emptyset$ ) và một nút  $r$  không thuộc  $T_1, T_2$ . Khi đó ta có thể tạo một cây nhị phân  $T$  với gốc là  $r$  và  $T_1$  là cây con bên trái đối với gốc  $r$  và  $T_2$  là cây con phải đối với gốc  $r$  của  $T$ .

Biểu diễn hình học của cây nhị phân cho trên hình 5.5.



Hình 5.5. Cây nhị phân

Cần phải hiểu rằng cây và cây nhị phân là hai khái niệm rất khác nhau: Cây (có gốc) không bao giờ rỗng, nó luôn luôn có ít nhất một đỉnh. Khi nó có nhiều hơn một đỉnh, mỗi đỉnh có thể không có cây con nào hoặc có thể có một hay nhiều cây con. Còn cây nhị phân có thể rỗng, khi nó không rỗng, thì mỗi đỉnh của CNP luôn luôn có 2 cây con.

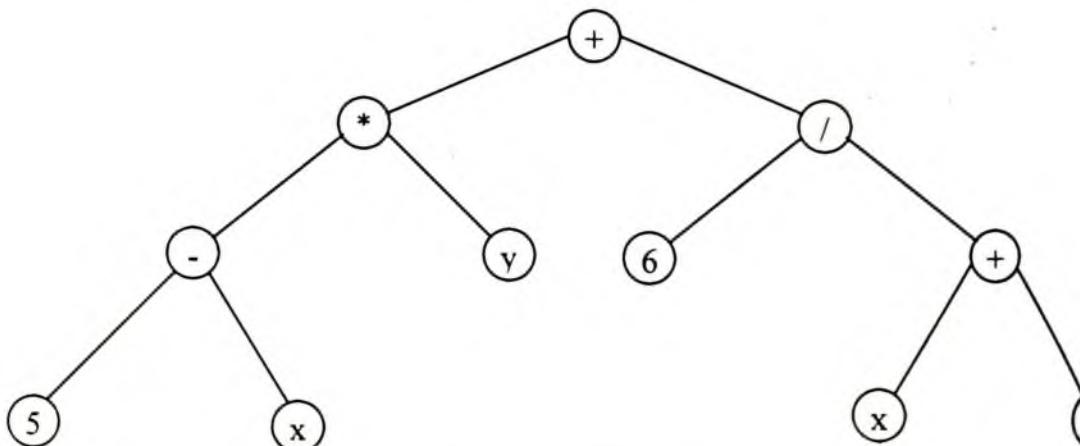
### 5.4.3. Khai báo cấu trúc CNP trong ngôn ngữ C

```
typedef struct Cnp
{
    DataType key;
    struct Cnp *pLeft, *pRight;
} T_Node;
typedef T_Node *np_Tree;
```

### 5.4.4. Một ứng dụng của CNP

a. **Cây biểu thức.** Một ứng dụng kinh điển của CNP là biểu diễn các biểu thức số học. Một biểu thức số học là một sự kết hợp có quy luật giữa các toán tử số học (+, -, \*, /,) với các toán hạng và các dấu ngoặc đơn để thay thế tính ưu tiên của các phép toán. Chẳng hạn  $(5-x)*y+6/(x+z)$ : các toán hạng là 5, x, y, 6, z; các toán tử là -, \*, +, /.

Mỗi biểu thức được biểu thị bằng một cây nhị phân duy nhất có cấu trúc được xác định bởi **tính ưu tiên** của các phép toán trong biểu thức. Một cây như thế gọi là *cây biểu thức*. Dưới đây là cây biểu thức của  $(5-x)*y+6/(x+z)$ .



Hình 5.6. Cây biểu thức

b. **Ba dạng biểu diễn Tiền tố, Trung tố, Hậu tố** (Các khái niệm Tiền tố, Trung tố, Hậu tố đã trình bày ở mục 4.4.1.c/c<sub>2</sub>, dưới đây nhắc lại ngắn gọn vì chúng và giải thích tại sao chúng có tên như vậy). Để trực quan do vậy dễ nhận thức, ta sẽ bàn về ba dạng này ngay trên biểu thức số học vừa nêu ở mục trên. Ba dạng biểu diễn ấy lần lượt có các tên gọi sau:

**Prefix (Tiền tố):**  $+* - 5xy / 6 + xz;$

**Infix (Trung tố):**  $5 - x * y + 6 / x + z;$

**Postfix (Hậu tố):**  $5x - y * 6xz + / +;$

Các dạng có tên như trên là do *cách thức duyệt ngang* cây quyết định:

- Nếu việc duyệt ngang cây theo *thứ tự trước* sẽ tạo ra biểu diễn tiền tố *Prefix*;

- Nếu việc duyệt ngang cây *không theo thứ tự* sẽ tạo ra biểu diễn nội (trung) tố *Infix*;

- Nếu việc duyệt ngang cây *theo thứ tự sau* sẽ tạo ra biểu diễn hậu tố *Postfix (RPN)*

#### 5.4.5. Các thao tác (Operations) trên CNP

- Khởi trị.
- Kiểm tra xem cây có rỗng không.
- Chèn thêm nút vào cây.
- Tìm một nút trên cây theo khóa cho trước.
- Xóa một nút theo khóa cho trước.
- Duyệt cây.

Vì cấu trúc của CNP không có những đặc tính nổi trội thuận tiện cho việc tra cứu, truy xuất, bô xung dữ liệu nên hầu như CNP ít được dùng trong thực tế. Bởi vậy ở đây chúng ta không xét tỷ mỷ việc cài đặt các thao tác trên CNP. Chúng ta dành phần này cho mục tìm hiểu *cây nhị phân tìm kiếm* là cấu trúc cây đặc biệt hơn CNP vì vậy nó rất thuận tiện cho truy xuất dữ liệu lưu trên nó nên nó thường được dùng trong thực tế.

### 5.5. CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE - BST)

#### 5.5.1. Mở đầu

Khi tìm kiếm hoặc truy-xuất các đối tượng được lưu trữ bởi các cấu trúc dữ liệu (trong đó có cây BST) ta quan tâm đến thuộc tính của chúng. Trong tin học các thuộc tính này gọi là *khóa (key)*. Ta có thể nói ngược lại: Khóa là thuộc tính của một lớp các đối tượng sao cho *hai đối tượng khác nhau phải có khóa khác nhau* (nếu ngược lại điều này thì ta không thể tìm kiếm các đối tượng được!).

Như khi học lập trình với một số ngôn ngữ như Pascal, C, Visual Basic... ta đã biết các đối tượng thường được lưu trữ bởi các cấu trúc (struct, còn gọi là

bản ghi) mà các *thành phần* (components) của struct *biểu diễn thuộc tính của đối tượng*.

Như vậy nếu dùng cây BST để lưu trữ và truy xuất dữ liệu thì mỗi nút của cây nhị phân nói chung và cây BST nói riêng sẽ được biểu diễn bởi một struct. Sau này thông tin gắn với mỗi nút trên cây BST chúng ta gọi là khóa của đối tượng tương ứng.

### 5.5.2. Định nghĩa đệ quy cây BST

Cây BST là *cây nhị phân* hoặc rỗng hoặc thỏa mãn các điều kiện sau:

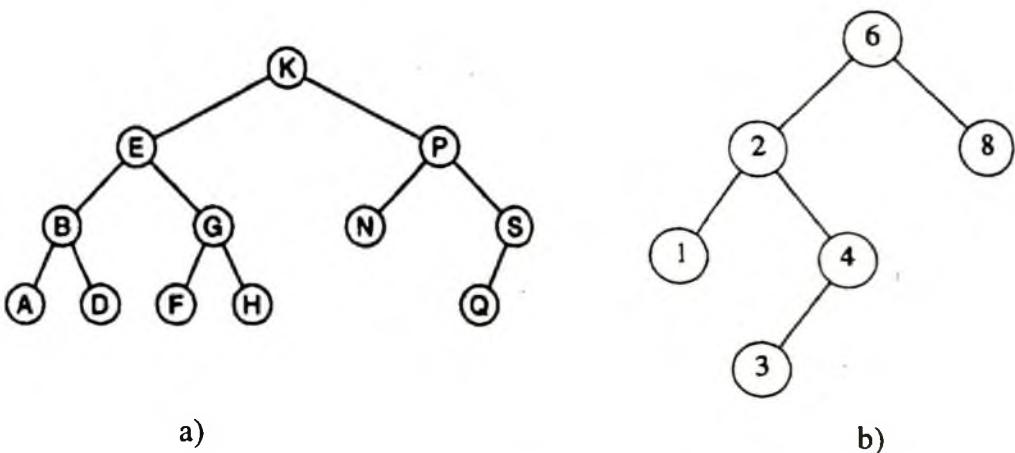
- Khóa của các nút *thuộc cây con trái nhỏ hơn* khóa của gốc.
- Khóa của gốc nhỏ hơn khóa của các nút thuộc cây con phải của gốc.
- Cây con trái và cây con phải của gốc cũng là cây BST.

### 5.5.3. Định nghĩa phi đệ quy cây BST

Cây BST là *cây nhị phân* mà khóa k tại một nút bất kỳ thỏa mãn điều kiện sau:

Giá trị của k ở nút bất kỳ  $\geq x$  với mọi khóa x ở các nút thuộc cây con bên trái nút đó (*nút hiện thời*) và  $k \leq y$  với mọi khóa y ở các nút thuộc cây con phải của nút hiện thời.

Hình 5.7 a), b) minh họa hình ảnh về cây BST trong hai tình huống cụ thể.



Hình 5.7. Cây nhị phân tìm kiếm

### 5.5.4. Các thao tác trên cây BST

#### a. Cài đặt các nút của BST

```
struct node
{
    DataType info;           //Trường info lưu dữ liệu có
                            //kiểu DataType
    node *trai, *phai;      //Khai báo con trỏ cây con
                            //trái và con trỏ cây con phải
};
```

#### b. Khởi trị cho BST

```
void khoitao(node *&ctgoc)
{
    ctgoc=NULL; } //Khởi đầu BST rỗng; ctgoc: con trỏ gốc
```

#### c. Tạo nút lá lưu nhãn x

```
node* taonut(int x)
{
    node* p = new node;
    p->info=x;
    p->trai=NULL;
    p->phai=NULL;
    return p;
};
```

#### d. Các thao tác duyệt BST

Khi tìm hiểu các danh sách tuyến tính trước đây chúng ta không quan tâm đến việc duyệt các phần tử của chúng vì bản chất cấu trúc tuyến tính của các danh sách này khiến cho việc duyệt các phần tử của chúng trở nên đơn giản tự nhiên. Vì cây nhị phân (BT) nói chung và cây nhị phân tìm kiếm nói riêng là một cấu trúc phi tuyến nên việc duyệt các phần tử của chúng là việc *quan trọng* với *tất cả các thao tác khác* trên BT hoặc BST và là việc *không đơn giản*.

Có ba phương thức duyệt BT hoặc BST dưới đây (để ngắn gọn sau này ta chỉ gọi là BST):

- *Duyệt theo thứ tự trước (Root-Left-Right: Gốc-Trái-Phải; viết tắt: RoLR)*

Cơ chế làm việc của cách duyệt này là: Trước tiên duyệt (thăm, xét) nút gốc, tiếp theo thăm các nút của cây con trái, sau cùng thăm các nút của cây con phải. (Một cách văn tắt: Gốc → Trái → Phải). Dưới đây là cài đặt đệ quy trên C của phương thức này. Ta đặt tên hàm mô tả thao tác này là `duyettruoc`.

```
void duyettruoc(node* ctgoc)
{
    if(ctgoc!=NULL)
    { printf("%3d", ctgoc->info);
        duyettruoc(ctgoc->trai);
        duyettruoc(ctgoc->phai);
    }
}
```

- *Duyệt theo thứ tự giữa (Left-Root-Right: Trái-Gốc-Phải, viết tắt: LRoR)*

Cơ chế làm việc của cách duyệt LRoR là: Trước nhất thăm các nút thuộc cây con trái, tiếp theo thăm gốc, sau cùng thăm các nút của cây con phải. (Nói gọn: Trái → Gốc → Phải). Cài đặt đệ quy của LRoR cho dưới đây (tên hàm mô tả phương thức duyệt này là `duyetgiua`).

```
void duyetgiua(node* ctgoc)
{
    if(ctgoc!=NULL)
    { duyetgiua(ctgoc->trai);
        printf("%3d", ctgoc->info);
        duyetgiua(ctgoc->phai);
    }
}
```

- *Duyệt theo thứ tự sau (Left-Right-Root: Trái-Phải-Gốc, viết tắt: LRRo)*

Cơ chế làm việc của cách duyệt LRRo: Đầu tiên thăm các nút của cây con trái, tiếp đến thăm các nút của cây con phải, cuối cùng thăm nút gốc. (Văn tắt: Trái → Phải → Gốc). Sau đây là cài đặt đệ quy trên C cho cách duyệt LRRo (tên hàm mô tả LRRo là `duyetsau`):

```

void duyetsau(node* ctgoc)
{
    if(ctgoc!=NULL)
    { duyetsau(ctgoc->trai);
    duyetsau(ctgoc->phai);
    printf("%3d",ctgoc->info);
    }
}

```

**e. Tìm đệ quy một nút có khóa x**

```

node* tim_dq(node* ctgoc,int x) //Cài đặt đệ quy cho
//thao tác này
{
    node* p;
    if(ctgoc==NULL) return NULL; //Nếu BST rỗng thì
//không tìm
    if(ctgoc->info==x) return ctgoc; //Nếu khóa x nằm
//ở gốc thì trả về địa chỉ gốc
    if(x<ctgoc->info) p=tim_dq(ctgoc->trai,x); //Nếu x
//ở các nút thuộc cây con trái thì tìm ở đó
    else p=tim_dq(ctgoc->phai,x); //Ngược lại tìm x ở các
//cây con phải
    return p;
}

```

**f. Tìm phi đệ quy một nút có khóa x**

```

node* tim_phi_dq(node *ctgoc,node *&fp,int x)
{ node* p=ctgoc;fp=NULL;
    while(p!=NULL)
    { if(x == p->info) return(p); // Nếu tìm thấy x
    //thì trả về địa chỉ của p (trỏ tới info)
        if(x < p->info) {fp=p;p = p->trai;}
        // Nếu x<dữ liệu cát ở trường info thì tìm ở các cây con trái
    }
}

```

```

        else
            { fp=p; p = p->phai; }           //Ngược lại, tìm x ở
        }
        return (NULL);                   //Không tìm thấy x thì
                                         trả về NULL
    }
}

```

#### g. Chèn một nút mới vào BST (Kiến tạo cây BST)

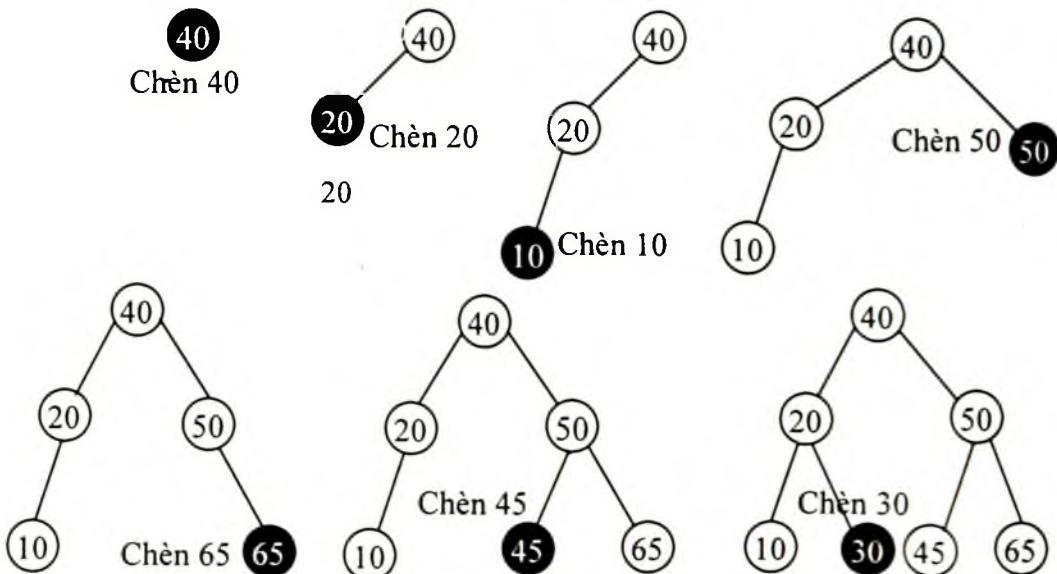
Cần hiểu rằng thao tác này không đơn giản như các thao tác vừa xét ở trên. Vì nếu không cẩn thận việc chèn một nút mới vào sẽ phá hủy các đặc tính của BST. Nghĩa là việc thêm một nút mới với khóa x luôn phải đảm bảo điều kiện ràng buộc của BST.

*Điễn đạt thuật toán chèn một khóa mới vào BST bằng ngôn ngữ tự nhiên:*

- Nếu cây *rỗng*, chèn khóa mới vào *gốc* của cây;
- Còn nếu khóa ở gốc trùng với khóa cần chèn thì không chèn (bỏ qua-skip);
- Còn nếu khóa mới *nhỏ hơn* khóa ở gốc thì chèn khóa mới vào *cây con trái*;
- Còn thì chèn khóa mới vào *cây con phải*.

+ *Ví dụ.* Cho danh sách khóa: 40, 20, 10 50, 65, 45, 30. Hãy tạo cây BST với bộ khóa ấy.

Dưới đây là sơ đồ mô tả cơ chế hoạt động của thuật toán chèn các khóa của bộ khóa đã cho vào BST. (kiến tạo cây BST):



Hình 5.8. Mô tả thuật toán chèn khóa mới vào cây BST (Kiến tạo cây)

*Diễn đạt thuật toán chèn bằng ngôn ngữ C:*

```

void chen_dq(node* &ctgoc, int x)
{
    if(ctgoc==NULL)           //Nếu cây rỗng thì tạo nút mới
        {ctgoc=taonut(x); return;}
    if(tim_dq(ctgoc,x))     //Nếu khóa cần chèn x đã có trên cây
        {printf("\n Nut da co, khong them
duoc!"); return;} //Thì không chèn thêm nữa
    if(x<ctgoc->info)
        chen_dq(ctgoc->trai,x); //Nếu x < dữ liệu ở các nút thuộc
                                //cây con trái thì chèn nó vào cây con trái
    else    chen_dq(ctgoc->phai,x); //Ngược lại chèn x vào
                                    //các cây con phải.
}

```

#### ***h. Xóa một nút trên BST***

Nếu chèn đã là thao tác không đơn giản như các thao tác kiểm tra xem cây có rỗng không, tạo sinh nút mới, tìm một khóa x trên cây..., thì thao tác xóa một nút trên cây còn phức tạp hơn nhiều. Lẽ thứ nhất, việc xóa một nút cũng rất dễ dẫn đến phá hoại tính chất của BST. Thứ hai là trước khi xóa phải phân biệt các trường hợp:

- + Nút cần xóa là nút lá
- + Nút cần xóa chỉ có một cây con (trái hoặc phải)
- + Nút cần xóa có hai cây con.

Cài đặt thao tác xóa là một việc khá khó với đa số sinh viên mới học môn này!

Sau đây là cài đặt trên C thao tác xóa một nút có khóa x.

```

void xoa(node *&ctgoc, int x)
{ node *fp,*p,*f,*rp,*q; //rp là nút thay thế cho nút p có
                           //khoá x, f là cha của nút thay thế rp
  p= tim_phi_dq(ctgoc,fp,x); //fp là nút cha của nút p

```

```
if (p==NULL) {printf("Khong tim thay nut x");return;}
```

#### //Nút lá

```
if (p->phai==NULL && p->trai==NULL) //Nếu không có
cây con trái và cây con phải thì nút ấy là lá
{if (p==ctgoc) {delete ctgoc; ctgoc=NULL;
return;} // Nếu nút cần xóa là gốc thì xóa vùng nhớ đã cấp cho con trỏ gốc
if (fp->trai==p) {fp->trai=NULL; delete p;
return;} // Nếu nút cần xóa là các nút ở cây con trái thì xóa vùng nhớ của
p trỏ tới cây con trái
if (fp->phai==p) {fp->phai=NULL; delete p;
return;} // Nếu nút cần xóa là các nút ở cây con phải
}; //thì xóa vùng nhớ của p trỏ tới cây con phải
```

#### //Nút chỉ có một cây con trái

```
if (p->trai!=NULL && p->phai==NULL)
{if (fp->trai==p) {fp->trai=p->trai;delete
p;return;}
if (fp->phai==p) {fp->phai=p->trai;delete
p;return;}
}
```

#### //Nút chỉ có một cây con phải

```
if (p->trai==NULL && p->phai!=NULL)
{if (fp->trai==p) {fp->trai=p->phai;delete p;return;}
if (fp->phai==p) {fp->phai=p->phai;delete p;return;}
};
```

#### //Nút p có 2 con

//Tim nút thay thế là nút phải nhất của cây con trái

```
f=p;
```

```
rp=p->trai;
```

```
while (rp->phai!=NULL) {f=rp; rp=rp->phai;}
```

```
f->phai=rp->trai; //rp là nút phải nhất, do đó không có con phải;
vì không con rp nên nút cha phải chỉ đến nút sau đó//
```

```

p->info=rp->info;
//Đổi nội dung của p và rp, rồi xóa rp
delete rp;
}

```

### 5.5.5. Cài đặt hoàn chỉnh Cây nhị phân tìm kiếm

- Những điều bạn đọc phải lưu ý trước khi Test chương trình:
  - + Trước tiên bao giờ cũng phải chọn mục 1 trong Menu (tức là phải gõ phím 1) để khởi trị BST;
  - + Tiếp theo chèn *dàn từng nút mới* vào cây (để tạo cây BST). Về thao tác này bạn có thể Test với bộ dữ liệu sau: 40, 20, 10, 50, 65, 45, 30
  - + Rồi bạn hãy chọn các thao tác tùy ý khác có trong Menu;

```

#include <conio.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <assert.h>

#define true 1
#define false 0

struct node {int info; node *trai, *phai;};
//=====

void khoitao(node *&ctgoc) //Khởi trị cho cây
{
    ctgoc=NULL;
}

//=====

int empty(node *ctgoc)
{
    return ctgoc==NULL; } //Kiểm tra xem cây có rỗng không
//=====

node* taonut(int x) //Tạo nút lá lưu nhãn x
{
    node* p = new node;
    p->info=x;
    p->trai=p->phai=NULL;
}

```

```
p->info=x;
p->trai=NULL;
p->phai=NULL;
return p;
};

//=====
//Duyệt cây theo thứ tự trước (RoLR): Gốc → Cây con Trái → Cây con Phải
void duyettruoc(node* ctgoc)
{
    if(ctgoc!=NULL)
    {
        printf("%5d", ctgoc->info);
        duyettruoc(ctgoc->trai);
        duyettruoc(ctgoc->phai);
    }
}

//=====
//Duyệt cây theo thứ tự giữa (LRoR): Cây con Trái → Gốc → Cây con Phải
void duyetgiua(node* ctgoc)
{
    if(ctgoc!=NULL)
    {
        duyetgiua(ctgoc->trai);
        printf("%5d", ctgoc->info);
        duyetgiua(ctgoc->phai);
    }
}

//=====
//Duyệt cây theo thứ tự sau (LRRo): Cây con Trái → Cây con Phải → Gốc
void duyetsau(node* ctgoc)
{
    if(ctgoc!=NULL)
    {
        duyetsau(ctgoc->trai);
        duyetsau(ctgoc->phai);
    }
}
```

```
    printf("%5d", ctgoc->info);
}
}

//=====

//Dùng đệ quy tìm nút có nhãn x
node* tim_dq(node* ctgoc, int x)
{
    node* p;
    if(ctgoc==NULL) return NULL;
    if(ctgoc->info==x) return ctgoc;
    if(x<ctgoc->info) p=tim_dq(ctgoc->trai,x);
    else p=tim_dq(ctgoc->phai,x);
    return p;
}

//=====

//Chèn nút mới (nhờ đệ quy) vào cây
void chen_dq(node* &ctgoc, int x)
{
    if(ctgoc==NULL) {ctgoc=taonut(x); return;}
    if(tim_dq(ctgoc,x))
        {printf("\n Nut da co, khong them duoc!"); return;}
    if(x<ctgoc->info) chen_dq(ctgoc->trai,x);
    else chen_dq(ctgoc->phai,x);
}

//=====

node* tim_phi_dq(node *ctgoc, node *&fp, int x)
//Tim Phi de quy nút có nhãn x
{
    node* p=ctgoc; fp=NULL;
    while(p!=NULL)
        {if(x == p->info) return(p); //tìm thấy
         if(x < p->info) {fp=p;p = p->trai;}}
```

```

    else
    {fp=p;p = p->phai;}
}
return(NULL);                                //Không tìm thấy
}

//=====
void xoa(node *&ctgoc, int x)
{node *fp,*p,*f,*rp,*q;                      //rp là nút thẻ mạng cho
nút p có nhãn x, f là nút cha của nút thẻ mạng rp
p=tim_phi_dq(ctgoc,fp,x);                     //fp là nút cha của nút p
if(p==NULL) {printf("Khong tim thay nut
x");return;}
//Nút muôn xóa là nút lá
if(p->phai==NULL && p->trai==NULL)
{if(p==ctgoc) {delete ctgoc;ctgoc=NULL;return;}
if(fp->trai==p) {fp->trai=NULL;delete p;return;}
if(fp->phai==p) {fp->phai=NULL;delete p;return;}
};

// Nút muôn xóa chỉ có một cây con trái
if(p->trai!=NULL && p->phai==NULL)
{if(fp->trai==p){fp->trai=p->trai;delete
p;return;}
if(fp->phai==p){fp->phai=p->trai;delete
p;return;}
}

// Nút muôn xóa chỉ có một cây con phải
if(p->trai==NULL && p->phai!=NULL)
{if(fp->trai==p){fp->trai=p->phai;delete p;return;}
if(fp->phai==p){fp->phai=p->phai;delete p;return;}
};

```

```
// Nút muốn xóa p có 2 nút con. Khi ấy, ta tìm nút thế mạng là nút phải nhất của cây con trái
f=p;
rp=p->trai;
while(rp->phai!=NULL) {f=rp; rp=rp->phai;}
f->phai=rp->trai; //rp: nút phải nhất, do đó không có con phải
p->info=rp->info; //Nạp dữ liệu ở trường info do rp trả về
                      //và trường info do p trả về
delete rp;           //rồi xóa rp
}
//=====
void main()
{clrscr();
int x,y;node* p, *ctgoc,*fp;
while(true)
{clrscr();
//Tạo menu
printf("1.Khoi tao.\n");
printf("2.Them nut chua khoa x.\n");
printf("3.Xoa nut co khoa x.\n");
printf("4.Tim nut co khoa x dung de quy.\n");
printf("5.Duyet cay theo thu tu truoc (NLR).\n");
printf("6.Duyet cay theo thu tu giua (LNR).\n");
printf("7.Duyet cay theo thu tu sau (LRN).\n");
printf("0.Ket thuc.\n");
printf("Hay nhap phim tu 1 -> z de chon:");
char chon=toupper(getch());
if(chon=='0') break;
switch(chon)
```

```
{case '1':khoitao(ctgoc);break;
    case '2':printf("\n Them nut co khoa x=");
                scanf("%d",&x);
                chen_dq(ctgoc,x);break;
    case '3':printf("\n Xoa nut co x=");
                scanf("%d",&x);
                xoa(ctgoc,x);break;
    case '4':printf("\n Tim nut co khoa
                    x=");scanf("%d",&x);
                p=tim_dq(ctgoc,x);if(p!=NULL)
                    {printf("\n Nut tim thay co khoa la:");
                     printf("%d",p->info);}
                    else printf("Khong tim thay.");break;
    case '5':if(!empty(ctgoc))
                {printf("\n Duyet theo thu tu TRUOC,");
                 printf(" Nut goc la:");
                 printf("%d\n",ctgoc->info);
                 }
                printf("Du lieu tai cac dinh cua cay:\n");
                duyettruoc(ctgoc);break;
    case '6':if(!empty(ctgoc))
                {printf("\nDuyet theo thu tu GIUA.");
                 printf(" Nut goc la:");
                 printf("%d\n",ctgoc->info);
                 }
                printf("Du lieu tai cac dinh cua cay:\n");
                duyetgiua(ctgoc);break;
    case '7':if(!empty(ctgoc))
                {printf("\nDuyet theo thu tu SAU.");
                 printf(" Nut goc la:");
                 printf("%d\n",ctgoc->info);}
```

```

    }
    printf("Gia tri khoa tai cac dinh cua
           cay:\n");
    duyetsau(ctgoc);break;
}
printf("\n\n Nhan phim bat ky de tiep tuc");
getch();
}
}

```

### 5.5.6. Độ phức tạp của các thao tác trên BST

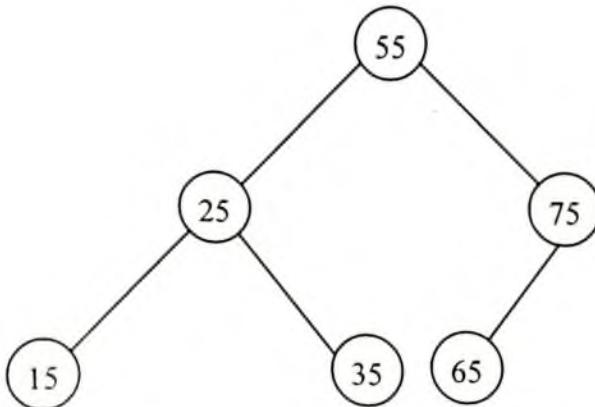
Độ phức tạp trung bình để thực hiện các thao tác trên BST là  $O(\log_2 n)$  với  $n$  là số nút của BST.

## 5.6. CÂY NHỊ PHÂN CÂN BẰNG (CÂY NPCB)

### 5.6.1. Cây NPCB hoàn toàn (cây NPCBHT)

**Định nghĩa:** Cây NPCBHT là *cây nhị phân tìm kiếm* mà tại mỗi nút của nó *số nút của cây con trái chênh lệch không quá một so với số nút của cây con phải*.

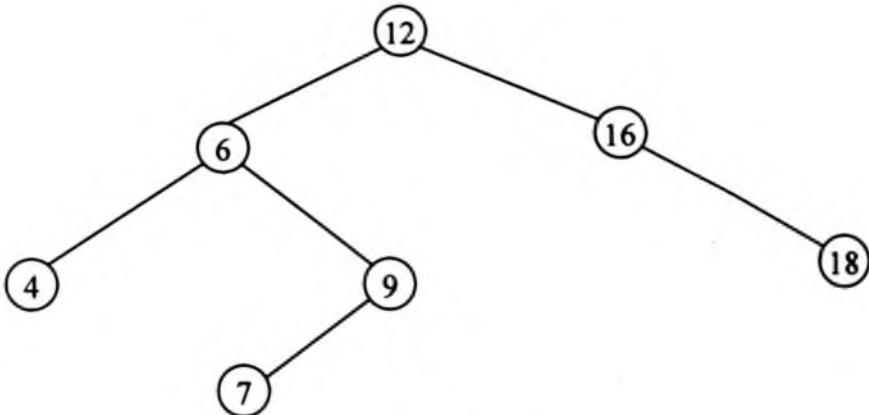
Một cây BST bất kỳ rất khó đạt trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng hoàn toàn khi chèn thêm hoặc hủy một nút trên nó. Thời gian khôi phục lại tính cân bằng hoàn toàn cũng rất lớn. Vì vậy trên thực tế cấu trúc cây NPCBHT ít được dùng. Hình 5.9 là một cây CBHT.



Hình 5.9. Cây nhị phân cân bằng hoàn toàn

### 5.6.2. Cây nhị phân cân bằng (Cây NPCB, tên khác: cây AVL)

a. **Định nghĩa:** Cây nhị phân cân bằng (CNPCB) là *cây nhị phân tìm kiếm* mà tại mỗi nút của nó **độ cao** của cây con trái và độ cao của cây con phải **chênh lệch không quá một**. Hình 5.10 là một CNPCB.



Hình 5.10. Cây AVL

Cây cân bằng hay cây AVL do hai nhà toán học Nga là G.M Adelsen Velskii và E.M Lendis nêu ra năm 1962.

#### b. Độ cao của cây AVL

Để xác định chiều cao của cây AVL trước tiên ta xét bài toán: Với số nút là n thì cây AVL phải có tối thiểu chiều cao là bao nhiêu?

Ta ký hiệu  $N(h)$  là số nút tối thiểu của cây AVL có chiều cao  $h$ .

Ta thấy  $N(0)=0$ ;  $N(1)=1$ ;  $N(2)=2$ .

Cây AVL tối thiểu sẽ có cây con AVL tối thiểu với chiều cao  $h-1$  và một cây con AVL tối thiểu với chiều cao  $h-2$ . Như vậy ta có:

$$N(h)=1+N(h-1)+N(h-2) \quad (*)$$

Ngoài ra  $N(h-1) > N(h-2)$  nên từ công thức (\*) suy ra:

$$N(h) > 2N(h-2)$$

$$N(h) > 2^2N(h-4)$$

.....

$$N(h) > 2^hN(h-2^h)$$

Suy ra:  $N(h) > 2^{h/2-1}$

và  $h < 2\log_2(N(h))+2$

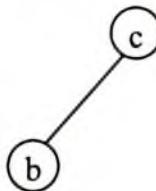
Như vậy, cây AVL có  $n$  nút thì chiều cao tối thiểu của nó là  $h=O(\log_2(n))$ .

c. **Các thao tác cơ bản trên cây AVL:** Trong số những thao tác cơ bản trên cây cây AVL, hai thao tác chèn thêm nút mới và xóa một nút luôn luôn phá hủy sự cân bằng của cây AVL. Do vậy ta cần xét các giải pháp khôi phục sự cân bằng của cây AVL. Các giải pháp này gọi là phép quay (Rotation). Thực chất của phép quay là *thay đổi vị trí các khoá ở các nút* sao cho *tính cân bằng của cây AVL được tái lập*. Có hai phép quay để tái lập cân bằng:

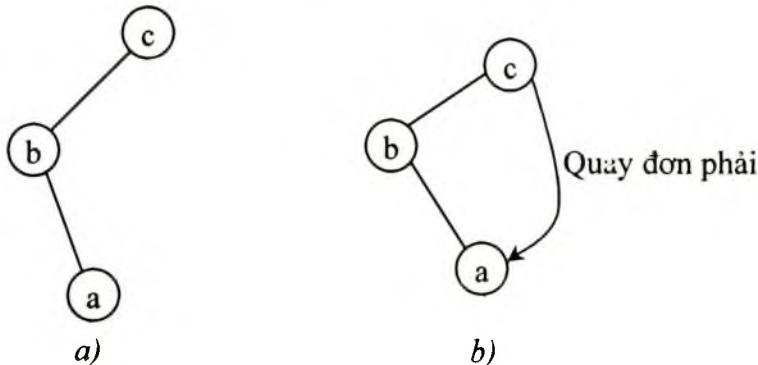
- + Quay đơn.
- + Quay đúp.

Và mỗi cách quay lại có quay “trái” và quay “phải”.

1. *Quay đơn phải (Single Right Rotation-SRRo)*. Chúng ta sẽ làm rõ khái niệm quay này bởi ví dụ đơn giản và độc đáo sau. Giả sử ban đầu chúng ta có cây như hình dưới đây (cây con phải rỗng):

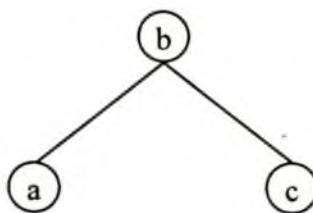


và bây giờ ta chèn nút a vào cây ấy (hình 5.11a):



Hình 5.11. Chèn nút mới và cây mất cân bằng (hình b)

Tính cân bằng của cây AVL bị vi phạm (hình 5.11 b): cây con trái có chiều cao = 2 nhưng cây con phải có chiều cao = 0 (*cây con phải rỗng!*). Như vậy độ vênh là 2, do đó ta phải quay đơn phải, xem hình 5.11 b): *Quay nút c theo chiều kim đồng hồ sang phải xuống dưới* được cây ở hình 5.12:

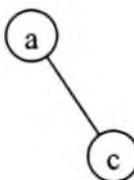


Hình 5.12. Cây đã tái lập cân bằng

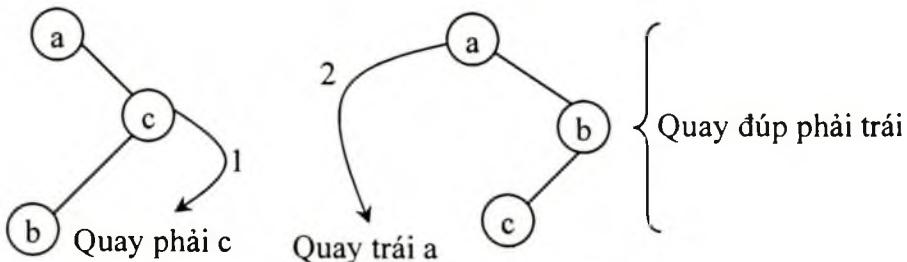
2. *Quay đơn trái* (Single Left Rotation - SLRo): Ngược lại với trường hợp trên (độc giả tự làm)

### 3. Quay đúp (Double Rotation)

Ta minh họa ví dụ quay đúp (double rotation). Mỗi kiểu quay đúp lại có: Double Right Rotation-DRRo: quay đúp phải, Double Left Rotation - DLRo quay đúp trái. Giả sử ta có cây sau đây (cây con trái rỗng).



Ta chèn nút b vào cây trên, cây sẽ mất cân bằng:



Lập lại cân bằng bằng cách: đầu tiên ta *quay phải* c xuống ta được cây "a-b-c", rồi ta *quay trái* (ngược chiều kim đồng hồ) nút a được cây cân bằng dưới đây:



Ta định lượng hóa khái niệm “cân bằng” của cây bằng cách đưa vào một hệ số gọi là *hệ số cân bằng (balance factor)* được định nghĩa bởi công thức sau:

$$\text{bal\_fac} = h_r - h_l$$

Trong đó  $h_r$  là chiều cao cây con phải;  $h_l$  là chiều cao cây con trái.

Như vậy: + Nếu  $\text{Bal\_fac} = 1$   $\left\{ \begin{array}{l} \text{Thì cây cân bằng} \\ \text{+ Hoặc } \text{Bal\_fac} = -1 \end{array} \right.$

Còn nếu + Nếu  $\text{Bal\_fac} = 2$   $\left\{ \begin{array}{l} \text{Cây mất cân bằng} \\ \text{+ Hoặc } \text{Bal\_fac} = -2 \end{array} \right.$

Tương tự độc giả tự xét trường hợp quay đúp trái phải (Double Left Rotation: DLRo)

#### 4. Khai báo hàng so sánh

Muốn phát hiện sự mất cân bằng của cây, trước hết ta phải so sánh chiều cao các nút của cây. Vì vậy đầu tiên ta cần khai báo một kiểu liệt kê dùng để chỉ thị kết quả so sánh như sau:

```
enum ss //Kiểu liệt kê so sánh cho ta kết quả so sánh độ cao các nút
{
    nh_h= -1, // nhỏ hơn
    bang=0, // bằng nhau
    lon_h=1, // lớn hơn
};
```

#### 5. Xây dựng lớp so sánh để so sánh chiều cao các nút

```
template <class KeyType>
class So_sanh //Lớp này thực hiện so sánh các nút chứa khóa KeyField
{
    private:
        KeyType KeyField; //KeyType là kiểu của trường lưu khóa KeyField
    public:
        So_sanh(KeyType key):KeyField(key) //key là
                                         tham biến hình thức. Hàm tạo và hàm hủy
    {};
```

```

ss Comp(KeyType key) const //So sánh một mục với
    //khóa đã cho và trả về kết quả
KeyType Key() const //Tạo khóa của một mục (Item)
{ return KeyField; } //Trả về giá trị (dữ liệu) lưu ở
                     //trường KeyField
};

```

*6. Mô phỏng cấu trúc node của AVL bởi C<sup>++</sup>*

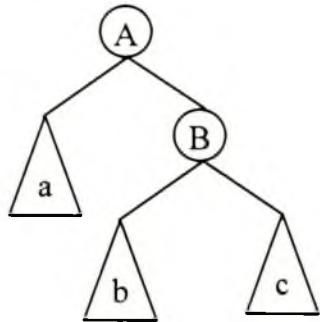
```

template <class KeyType>
class AvlNode
{
private:
    So_sanh<KeyType> *pData      // Con trỏ trường dữ liệu
    AvlNode<KeyType> *tree;        // Con trỏ cây
    AvlNode<KeyType> *pSubtree[2]; // Con trỏ cây con
    short   Bal_fac;             // Hệ số cân bằng
};

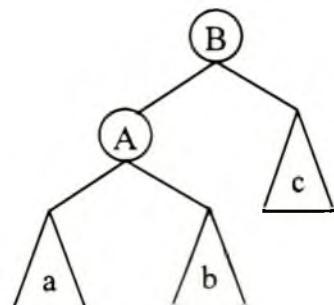
```

*7. Tính toán cân bằng mới sau quay*

*a. Để tính sự cân bằng mới sau một phép quay đơn trái ta xét trường hợp sau (hình 5.13):*



Trước khi quay



Sau khi quay

*Hình 5.13. Trước quay và sau quay (để bảo đảm đặc tính của cây AVL)*

Cây ở hình bên trái là cây trước khi quay, cây ở hình bên phải là cây sau khi quay. *Chữ cái hoa để đánh dấu các nút, còn chữ cái thường biểu thị cây con.*

Ta tính toán sự cân bằng mới của các nút A và B như sau:

$$\text{NewBal}(A) = \text{ht}(b) - \text{ht}(a) \quad // \text{Newbal: cân bằng mới; ht() là hàm chiều cao}$$

$$\text{OldBal}(A) = \text{ht}(B) - \text{ht}(a) = (1 + \max(\text{ht}(b), \text{ht}(c))) - \text{ht}(a) \quad // \text{OldBal: cân bằng cũ (cân bằng lúc trước)}$$

Trừ vế với vế ở hai công thức trên ta được:

$$\text{NewBal}(A) - \text{OldBal}(A) = \text{ht}(b) - (1 + \max(\text{ht}(b), \text{ht}(c))) + \text{ht}(a) - \text{ht}(a)$$

Đến đây ta có:

$$\text{NewBal}(A) = \text{OldBal}(A) - 1 - (\max(\text{ht}(b), \text{ht}(c)) - \text{ht}(b))$$

Biết rằng:  $\max(x, y) - z = \max(x-z, y-z)$ , nên ta được:

$$\text{NewBal}(A) = \text{OldBal}(A) - 1 - (\max(\text{ht}(b) - \text{ht}(b), \text{ht}(c) - \text{ht}(b)))$$

Song  $\text{ht}(c) - \text{ht}(b)$  là  $\text{OldBal}(B)$  vì vậy có:

$$\text{NewBal}(A) = \text{OldBal}(A) - 1 - (\max(0, \text{OldBal}(B))) = \text{OldBal}(A) - 1 - \max(0, \text{OldBal}(B))$$

*Vậy đổi với nút A ta có:*

$$\text{NewBal}(A) = \text{OldBal}(A) - 1 - \max(0, \text{OldBal}(B))$$

*Tính toán tương tự với nút B có:*

$$\text{NewBal}(B) = \text{ht}(c) - \text{ht}(A) = \text{ht}(c) - (1 + \max(\text{ht}(a), \text{ht}(b)))$$

$$\text{OldBal}(B) = \text{ht}(c) - \text{ht}(b)$$

Trừ đẳng thức 2 vào đẳng thức 1 ta được:

$$\text{NewBal}(B) - \text{OldBal}(B) = \text{ht}(c) - \text{ht}(c) + \text{ht}(b) - (1 + \max(\text{ht}(a), \text{ht}(b)))$$

Thực hiện một biến đổi sơ cấp (như đã làm với nút A) ta có:

$$\text{NewBal}(B) = \text{OldBal}(B) - 1 - (\max(\text{ht}(a), \text{ht}(b)) - \text{ht}(b)) = \text{OldBal}(B) - 1 - (\max(\text{ht}(a) - \text{ht}(b), \text{ht}(b) - \text{ht}(b)))$$

Mà  $\text{ht}(a) - \text{ht}(b)$  is  $-(\text{ht}(b) - \text{ht}(a)) = -\text{NewBal}(A)$ , vì vậy:

$$\text{NewBal}(B) = \text{OldBal}(B) - 1 - \max(-\text{NewBal}(A), 0)$$

Sử dụng hệ thức  $\min(x, y) = -\max(-x, -y)$  ta có:

$$\text{NewBal}(B) = \text{OldBal}(B) - 1 + \min(\text{NewBal}(A), 0)$$

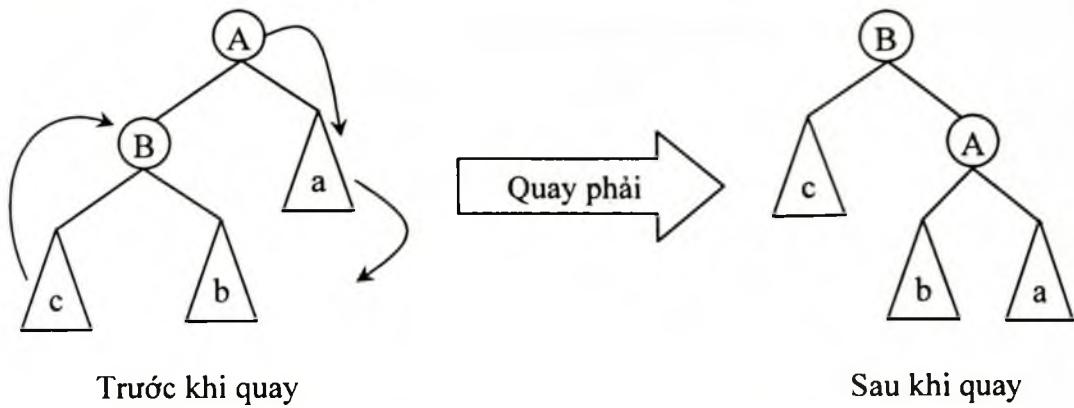
Như vậy, sau phép quay đơn trái ta có sự cân bằng mới của các nút A, B được đánh giá bởi hai công thức sau:

$$\text{NewBal}(A) = \text{OldBal}(A) - 1 - \max(\text{OldBal}(B), 0)$$

$$\text{NewBal}(B) = \text{OldBal}(B) - 1 + \min(\text{NewBal}(A), 0)$$

b. Tính toán sự cân bằng mới sau quay đơn phải:

Ta xét hình dưới đây, nó như là một ảnh gương của trường hợp a) đã xét ở trên. Nghĩa là trong trường hợp này trước tiên ta quay phải nút A và cây con a xuống dưới, quay phải cây con c lên trên (xem hình bên dưới).



Làm tương tự như trường hợp quay trái, ta sẽ có công thức tính cân bằng mới cho *trường hợp quay phải* như dưới đây:

$$\text{NewBal}(A) = \text{OldBal}(A) + 1 - \min(\text{OldBal}(B), 0)$$

$$\text{NewBal}(B) = \text{OldBal}(B) + 1 + \max(\text{NewBal}(A), 0)$$

// Ký hiệu cây con trái và cây con phải bởi hai hàng kiểu liệt kê:

```
enum Subtree { LEFT = 0, RIGHT = 1 };

inline int
MIN(int a, int b) { return (a < b) ? a : b; }

inline int
MAX(int a, int b) { return (a > b) ? a : b; }
```

*Đến đây ta có code trên C++ cho thao tác quay đơn trái và quay đơn phải như sau:*

// Mô tả thuật toán quay đơn trái

```
template <class KeyType>
void AvlNode<KeyType>::RotateLeft(AvlNode<KeyType>
* & root)
{
    AvlNode<KeyType> *oldRoot = root;
```

```

//Mô tả quay đơn trái
root = root->pSubtree[RIGHT];
oldRoot->pSubtree[RIGHT] = root->pSubtree[LEFT];
root->pSubtree[LEFT] = oldRoot;

// Cập nhật cân bằng
oldRoot->Bal_fac -= (1 + MAX(root->Bal_fac, 0));
root->Bal_fac -= (1 - MIN(oldRoot->Bal_fac, 0));
}

//Mô tả thuật toán quay đơn phải
template <class KeyType>
void AvlNode<KeyType>::RotaRight(AvlNode<KeyType>
                                     * & root)
{
    AvlNode<KeyType> * oldRoot = root;

    // Mô tả quay đơn phải
    root = root->pSubtree[LEFT];
    oldRoot->pSubtree[LEFT] = root->pSubtree[RIGHT];
    root->pSubtree[RIGHT] = oldRoot;

    // Cập nhật sự cân bằng
    oldRoot->Bal_fac += (1 - MIN(root->Bal_fac, 0));
    root->Bal_fac += (1 + MAX(oldRoot->Bal_fac, 0));
}

```

Ta có thể làm cho mã trên gọn hơn bởi cách chỉ dùng một phép quay. Để làm việc này ta thêm vào một tham số gọi là hướng quay. Khi đó ta không quan tâm đến LEFT hay RIGHT nữa mà chỉ cần quan tâm đến các hiệu số sau đổi với phép quay:

$$\text{RIGHT}=1-\text{LEFT} \text{ và } \text{LEFT}=1-\text{RIGHT}$$

Như vậy một trong hai cây con trái là 0 thì cây kia là 1 và ngược lại.

Sử dụng tham số mới (hướng) vừa nêu trên đây, ta được code trên C++ cho thao tác quay đơn là:

```

        inline Subtree
        Opposite(Subtree Tr){ return Subtree(1 - int(Tr));
    }

    // Đây là phép quay đơn - Nó là hàm thành viên tĩnh mô tả sự quay cho
    // một hướng xác định

    template <class KeyType>
    void
AvlNode<KeyType>::RotateOnce(AvlNode<KeyType> * &
                               root, Subtree Tr)

{
    AvlNode<KeyType> * oldRoot = root;
    Subtree otherTr = Opposite(Tr);

    // Quay
    root = tree->pSubtree[otherTr];
    oldRoot->pSubtree[otherTr] = tree->pSubtree[Tr];
    root->pSubtree[Tr] = oldRoot;

    // Cập nhật cân bằng
    if (Tr == LEFT)
    {
        oldRoot->Bal_fac -= (1 + MAX(root->Bal_fac, 0));
        root->Bal_fac -= (1 - MIN(oldRoot->Bal_fac, 0));
    }
    else /* Tr == RIGHT */
    {
        oldRoot->Bal_fac += (1 - MIN(root->Bal_fac, 0));
        root->Bal_fac += (1 + MAX(oldRoot->Bal_fac, 0));
    }
}

```

Ta vẫn có thể làm gọn mã trên hơn nữa bằng cách sử dụng quay vòng các công thức cập nhật cân bằng và để ý rằng:  $\max(x, y) = -\min(-x, -y)$

<i>Cho quay trái</i>
oldRoot->Bal_fac -= (1 + MAX(tree->Bal_fac, 0));
tree->Bal_fac -= (1 - MIN(oldRoot->Bal_fac, 0));
<i>Cho quay phải</i>
oldRoot->Bal_fac += (1 - MIN(tree->Bal_fac, 0));
tree->Bal_fac += (1 + MAX(oldRoot->Bal_fac, 0));

Ta dùng luật “max\_min” ở trên thì các công thức cập nhật cân bằng trên đây trở thành:

<i>Cho quay trái</i>
oldRoot->Bal_fac -= (1 + MAX(+(tree->Bal_fac), 0));
tree->Bal_fac -= (1 + MAX(-(oldRoot->Bal_fac), 0));
<i>Cho quay phải</i>
oldRoot->Bal_fac += (1 + MAX(-(tree->Bal_fac), 0));
tree->Bal_fac += (1 + MAX(+(oldRoot->Bal_fac), 0));

Đến đây chú ý rằng sự khác biệt giữa việc cập nhật cân bằng của quay phải và quay trái chỉ là dấu + hay dấu -.

Ta tiếp tục “nghiên cứu” với mục đích sao cho code có thể thuận tiện và càng gọn nữa càng tốt. Ta xây dựng một hàm có thể ánh xạ LEFT với +1 và RIGHT với -1 thì có thể code sẽ tiện dụng hơn. Một hàm như vậy là:  $f(x) = 1 - 2x$ . Hàm này sẽ ánh xạ 0 với 1 (khi  $x=0$ ) và 1 với -1 (khi  $x=1$ ); bây giờ ta thử nhân  $f(x)$  với hiệu RIGHT-LEFT (và đặt tích này là  $g(x)$ ):

$$g(x) = (1 - 2x)(RIGHT - LEFT)$$

+ Nếu LEFT = 0 và RIGHT = 1 thì:

$$g(LEFT) = (1 - 2*0)(1 - 0) = 1*1 = 1$$

$$g(RIGHT) = (1 - 2*1)(1 - 0) = (-1)*1 = -1$$

+ Nếu LEFT = 1 và RIGHT = 0 thì:

$$g(LEFT) = (1 - 2*1)(0 - 1) = (-1)*(-1) = 1$$

$$g(RIGHT) = (1 - 2*0)(0 - 1) = 1*(-1) = -1$$

Như vậy hàm  $g$  ánh xạ LEFT với +1 và RIGHT với -1

Bây giờ ta đưa vào một biến mới gọi là hs (hệ số) để gán cho giá trị  $g(Tr)$  thì ta có thể cập nhật sự cân bằng của phép quay mà không cần dùng lệnh điều kiện:

**Quay trên hướng Tr**

```
oldRoot->Bal_fac -= hs * (1 + MAX(hs * tree->Bal_fac, 0));
tree->Bal_fac += hs * (1 + MAX(hs * oldRoot->Bal_fac, 0));
```

Dùng kết quả này, code mô tả phép quay bây giờ trở thành:

// RotateOnce: Hàm thành viên tĩnh sẽ biểu diễn phép quay đơn theo một hướng đã cho

// Hàm này trả về 1 nếu chiều cao của cây thay đổi trong khi quay, ngược lại nó trả về 0

```
template <class KeyType>
void
AvlNode<KeyType>::RotateOnce (AvlNode<KeyType> * &
root, Subtree Tr)
{
    AvlNode<KeyType> * oldRoot = root;
    Subtree otherTr = Opposite(Tr);
    short factor = (RIGHT - LEFT) * (1 - (2 * Tr));
    // rotate
    root = tree->pSubtree[otherTr];
    oldRoot->pSubtree[otherTr] = tree->pSubtree[Tr];
    root->pSubtree[Tr] = oldRoot;
    // update balances
    oldRoot->Bal_fac -= hs * (1 + MAX(hs * root-
                                         >Bal_fac, 0));
    root->Bal_fac += hs * (1 + MAX(factor *
                                         oldRoot->Bal_fac, 0));
}
```

Thế là ta được phiên bản gọn “nhất” của phép quay và khi kiểm thử, nó không yêu cầu điều kiện thử (nói rõ hơn: không cần các lệnh điều kiện)

c. *Hàm mô tả quay đúp cho dưới đây:*

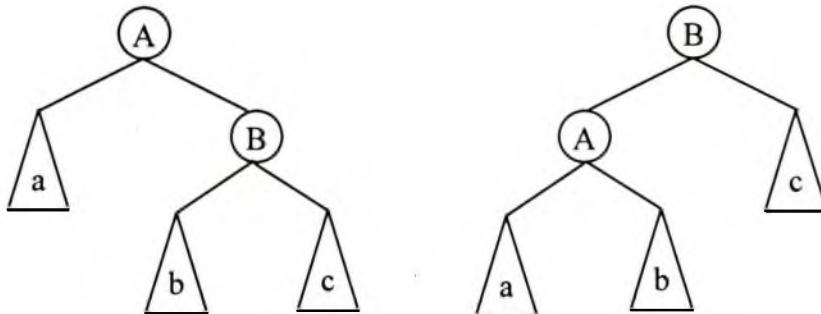
// RotateTwice - Hàm thành viên tĩnh mô tả phép quay một nút theo hướng đã cho rồi theo hướng ngược lại để cân bằng cây AVL

//Hàm này trả về 1 nếu chiều cao cây thay đổi trong khi quay, ngược lại nó trả về 0

```
template <class KeyType>
void
AvlNode<KeyType>::RotateTwice(AvlNode<KeyType> * &
                                root, Subtree Tr)
{
    Subtree otherTr = Opposite(Tr);
    RotateOnce(root->pSubtree[otherTr], otherTr);
    RotateOnce(root, Tr);
}
```

### 8. Một phương pháp khác tính toán sự cân bằng mới sau khi quay

Ta có thể dùng phương pháp khác có vẻ đơn giản hơn cách trình bày ở trên để tính toán cân bằng mới. Phương pháp này chỉ dựa vào một khái niệm rất đơn giản là “chiều cao (đã) thay đổi” hay “chiều cao không thay đổi”. Chẳng hạn quay đơn trái ta có một trong hai khả năng sau:



Hệ số cân bằng				
	Trước quay		Sau quay	
Tình huống 1	A = +2	B = +1	A = 0	B = 0
Tình huống 2	A = +2	B = 0	A = +1	B = -1

Như vậy cả hai tình huống ta đều có  $NewB = OldB - 1$  và  $newA = -newB$  do đó  $A = -(-B)$  đối với quay đơn trái).

Đối với quay đơn phải thì khả năng cho ở bảng dưới đây: (Sơ đồ của trường hợp này là ảnh gương của trường hợp quay trái ở trên- xem bảng dưới đây):

Hệ số cân bằng				
	Trước quay		Sau quay	
Tình huống 1	A = -2	B = -1	A = 0	B = 0
Tình huống 2	A = -2	B = 0	A = -1	B = -1

Vậy cả hai tình huống đều có NewB = OldB +1 và newA = -newB như vậy ta có A = - (++B) đối với quay đơn phải.

Lợi dụng các nhận xét trên đây ta có hàm mô tả quay đơn như dưới đây sau khi định nghĩa hai hằng kiểu liệt kê có tên là HEIGHT\_NOCHANGE và HEIGHT\_CHANGE

```

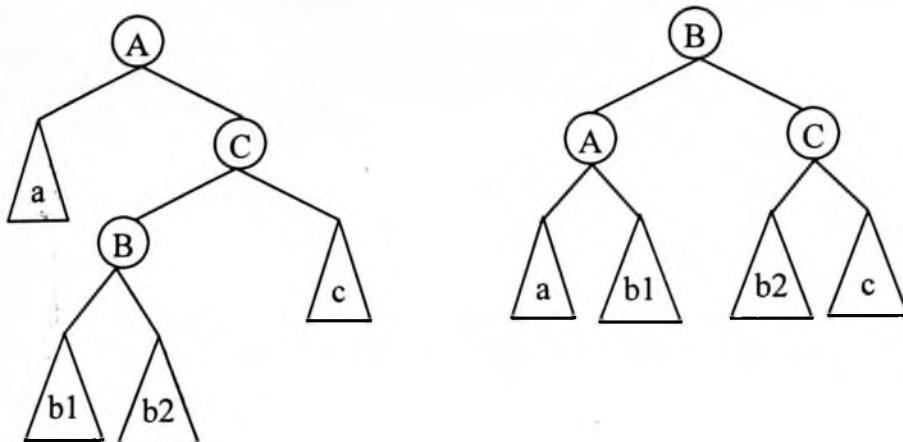
// Dùng hằng kiểu liệt kê chỉ thị cho sự thay đổi chiều cao
enum Do_cao{HEIGHT_NOCHANGE = 0, HEIGHT_CHANGE=1};

// Hàm RotateOnce - Là một hàm thành viên mô tả phép quay đơn theo
// một hướng đã chọn dựa vào 2 hằng định nghĩa ở trên

//Hàm trả về 1 nếu chiều cao cây thay đổi trong khi quay, và nó trả về 0
//trong trường hợp ngược lại
template <class KeyType>
int
AvlNode<KeyType>::RotateOnce(AvlNode<KeyType> * &
                               root, Subtree Tr)
{
    Subtree otherTr = Opposite(Tr);
    AvlNode<KeyType> * oldRoot = root;
    int heightChange = (root->pSubtree[otherTr]->Bal_fac == 0)? HEIGHT_NOCHANGE:HEIGHT_CHANGE;
    //Gán gốc mới
    root = oldRoot->pSubtree[otherTr];
    //Gốc mới thay đổi cây con "Tr" đối với nút cha của nó
    oldRoot->pSubtree[otherTr] = root->pSubtree[Tr];
    root->pSubtree[dir] = oldRoot;
    // Cập nhật cân bằng
    oldRoot->Bal_fac = -((Tr == LEFT) ? --(root->Bal_fac) : ++(root->Bal_fac));
    return heightChange;
}

```

Đối với quay đúp Trái - Phải (LR) ta có một trong hai khả năng sau:



Hệ số cân bằng						
	Trước quay			Sau quay		
case 1:	A = +2	B = +1	C = -1	A = -1	B = 0	C = 0
case 2:	A = +2	B = 0	C = -1	A = 0	B = 0	C = 0
case 3:	A = +2	B = -1	C = -1	A = 0	B = 0	C = +1

Như vậy trong mọi tình huống ta đều có

$$\text{newA} = -\max(\text{oldB}, 0)$$

$$\text{newC} = -\min(\text{oldB}, 0)$$

$$\text{newB} = 0$$

Còn đối với quay đúp Phải-Trái (DRRo) ta có các khả năng sau (trường hợp này là ảnh gương của trường hợp Trái-Phải)

Hệ số cân bằng						
	Trước quay			Sau quay		
Trường hợp 1:	A = -2	B = +1	C = +1	A = 0	B = 0	C = -1
Trường hợp 2:	A = -2	B = 0	C = +1	A = 0	B = 0	C = 0
Trường hợp 3:	A = -2	B = -1	C = +1	A = +1	B = 0	C = 0

Như vậy trong mọi trường hợp ta đều có:

$$\text{newA} = -\min(\text{oldB}, 0)$$

$$\text{newC} = -\max(\text{oldB}, 0)$$

$$\text{newB} = 0$$

Nhìn hai bảng trên, rõ ràng phép quay đúp DRRo là ảnh gương của DLRo. Nói rõ hơn: các nút A và C trong phép quay mới (DRRo) là kết quả của sự thay đổi đơn giản hệ số cân bằng ở phép quay ban đầu (DLRo). Điều này có nghĩa là hệ số cân bằng mới của cây con trái cũng như hệ số cân bằng mới của cây con phải thỏa mãn các hệ thức sau:

```
new(left) = -max(oldB, 0)
new(right) = -min(oldB, 0)
new(root) = 0
```

*Đến đây code của phép quay đúp được viết như sau:*

// RotateTwice: Là hàm thành viên quay đúp nút cho trước theo hướng đã định để khôi phục cân bằng cho cây AVL

// Hàm này trả về 1 nếu chiều cao của cây thay đổi trong khi quay, ngược lại nó trả về 0

```
template <class KeyType>
{
    int
    AvlNode<KeyType>::RotateTwice(AvlNode<KeyType>
                                    * & root, Subtree Tr)

    {
        Subtree otherTr = Opposite(Tr);
        AvlNode<KeyType> *oldRoot = root;
        AvlNode<KeyType> *oldOtherTr = root-
                                >pSubtree[otherTr];

        // Gán gốc mới
        root = oldRoot->pSubtree[otherTr]-
               >pSubtree[Tr];

        // Gốc mới thay đổi cây con "Tr" của nó đối với nút ông của nó
        // (grandparent)
        oldRoot->pSubtree[otherTr] = root->pSubtree[Tr];
        root->pSubtree[Tr] = oldRoot;

        // Gốc mới thay đổi cây con "otherTr" của nó đối với nút cha của nó
        // (parent)
    }
}
```

```

oldOtherTr->pSubtree[Tr] = root->pSubtree[otherTr];
root->pSubtree[otherTr] = oldOtherTr;
// Cập nhật cân bằng
root->pSubtree[LEFT]->Bal_fac = -MAX(root-
                                             >Bal_fac, 0);
root->pSubtree[RIGHT]->Bal_fac = -MIN(root-
                                              >Bal_fac, 0);
root->Bal_fac = 0;
return HEIGHT_CHANGE; // Phép quay đúp luôn luôn rút ngắn
                      toàn bộ chiều cao của cây
}

```

Bây giờ ta có thể viết ngay một hàm để mỗi khi gặp cây con có chiều cao quá lớn ta sẽ gọi nó để lập lại cân bằng cho cây. Trước hết ta khai báo các hằng kiểu liệt kê như dưới đây:

```

enum balance_t { LEFT_HEAVY = -1, BALANCED = 0,
                  RIGHT_HEAVY = 1 };

// Dưới đây là hàm LEFT_IMBALANCE(short bal). Mỗi khi cây quá cao
// ở phần bên trái thì nó trả về true
inline static int
LEFT_IMBALANCE(short bal) { return (bal < LEFT_HEAVY); }

// và hàm RIGHT_IMBALANCE(short bal). Mỗi khi cây quá cao ở phần
// bên phải thì nó trả về true
inline static int
RIGHT_IMBALANCE(short bal) { return (bal < RIGHT_HEAVY); }

// Tiếp theo là hàm khôi phục lại cân bằng Rebalance
// Rebalance - hàm thành viên này lập lại cân bằng cho cây con (mỗi khi
// nó mất cân bằng)

// Nó trả về 1 nếu chiều cao cây thay đổi trong khi quay, ngược lại nó trả về 0
template <class KeyType>
int AvlNode<KeyType>::ReBalance(AvlNode<KeyType>
                                    * & root)

```

```

{
    int heightChange = HEIGHT_NOCHANGE;
    if (LEFT_IMBALANCE(root->Bal_fac))
        {if (root->pSubtree[LEFT]->Bal_fac == RIGHT_HEAVY)
         {
// Quay đúp phải (DRRo)
            heightChange = RotateTwice(root, RIGHT);
        }
        else
        {
// Quay đơn phải SRRo
            heightChange = RotateOnce(root, RIGHT);
        }
    }
    else if (RIGHT_IMBALANCE(root->Bal_fac))
    {
        if (root->pSubtree[RIGHT]->Bal_fac == LEFT_HEAVY)
        {
// Cần phép quay đúp trái DLRo
            heightChange = RotateTwice(root, LEFT);
        }
        else
        {
// Cần phép quay đơn trái SLRo
            heightChange = RotateOnce(root, LEFT);
        }
    }
    return heightChange;
}

```

### 9. Xác định khi nào thì chiều cao của cây con hiện hành thay đổi

Trước khi viết code mô tả thuật toán chèn và xóa một nút ta cần xây dựng hàm xác định khi nào chiều cao cây hiện hành thay đổi với việc sử dụng kiểu liệt kê so sánh đã khai báo ở trên:

// Ta nhắc lại các hằng này như sau:

```
enum ss
{
    nh_h= -1,      // nhỏ hơn
    bang=0,        // bằng nhau
    lon_h=1,       // lớn hơn
};

template <class KeyType>
ssAvlNode<KeyType>::Comp (KeyType key, sscmp) const
{
    switch (cmp)
    {
        case nh_h : return pData->Comp(key); break;
        case bang : return (pSubtree[LEFT] ==
NULL) ? bang : nh_h; //Tìm mục data nhỏ nhất
        case lon_h : return (pSubtree[RIGHT] ==
NULL) ? bang : lon_h; //Tìm mục data lớn nhất
    }
}
```

### 10. Thao tác chèn nút mới vào cây AVL (mô tả bằng C++)

// Insert - hàm này chèn khóa mới vào cây. Nếu khóa đã tồn tại thì không chèn nữa và hàm trả về return

// Ngược lại hàm trả về NULL → chỉ thị (indicate) khóa đã chèn thành công.

```
template <class KeyType>
So_sanh<KeyType>
*AvlNode<KeyType>::Insert (So_sanh<KeyType> * item,
AvlNode<KeyType> *& root, int & change)
{
```

```

// Kiểm tra xem cây có rỗng không
if (root == NULL)
{
    // Chèn nút mới ở đây
    root = new AvlNode<KeyType>(item);
    change = HEIGHT_CHANGE;
    return NULL;
}

// Khởi tạo
So_sanh<KeyType> * found = NULL;
int increase = 0;
// So sánh các mục và xác định hướng tìm kiếm
ss result = root->Comp(item->Key());
Subtree Tr = (result == nh_h) ? LEFT : RIGHT;
if (result != bang)
{
    // Chèn vào cây con "Tr"
    found = Insert(item, root->pSubtree[dir],
                    change);
    if (found) return found;
    increase = result * change;// tăng hệ số cân bằng
}
else
{
    // khóa đã tồn tại ở cây này (nên không chèn)
    increase = HEIGHT_NOCHANGE;// và vì thế chiều cao
                                // cây không thay đổi
    return root->pData;
}
root->Bal_fac += increase; // Cập nhật hệ số cân bằng

```

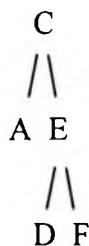
```

change = (increase && root->Bal_fac) ? (1 -
    ReBalance(root)) : HEIGHT_NOCHANGE;
return NULL;
}

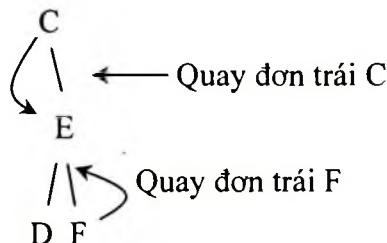
```

### 11. Thao tác xoá một nút (Mô tả trên C++):

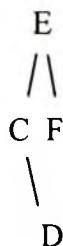
Thao tác này phiền phức hơn thao tác chèn nhiều. Ta có một cây như sau:



Xoá nút A sinh ra cây mất cân bằng sau: Cây con trái lúc này rỗng, cây con phải có chiều cao là 2)



Kiểu mất cân bằng này không có trong chèn, chỉ có trong thao tác xóa! Cây con trái lúc này rỗng, cây con phải có chiều cao là 2. Hay nói khác đi: gốc của cây hiện hành có chiều cao là 2 trong khi cây con có chiều cao bằng 0. Thực hiện phép quay trái để lập lại cân bằng ta được:



Cây này có chiều cao giống hệt chiều cao của cây trước lúc quay. Đổi với phép chèn ta chỉ cần xác định xem chiều cao cây con có bị thay đổi không. Song đổi với phép xóa ngoài việc ấy ta còn phải kiểm tra toàn bộ các nút phía trên của cây.



```

decrease = result * change; //Ghi nhớ hệ số cân bằng giảm
}
else
{
    // Tìm thấy khóa
    found = root->pData;
    // Có các trường hợp sau:
    // 1) Nút cần xóa là lá: Xóa nó
    // 2) Nút là nhánh chỉ có 1 con. Tạo con trỏ “gốc” trả về con này rồi xóa
    // 3) Nút có hai con, tráo đổi các mục với hậu thế của “gốc” (mục nhỏ hơn
        ở cây con phải) và xóa hậu thế từ cây con phải của “gốc”.
    // -----
    if ((root->pSubtree[LEFT] == NULL) && (root-
        >pSubtree[RIGHT] == NULL))
    {
        // Ta có nút lá: xóa nó đi
        delete root;
        root = NULL;
        change = HEIGHT_CHANGE; // Chiều cao thay đổi từ 1
                                // xuống 0
        return found;
    }
    else if ((root->pSubtree[LEFT] == NULL) ||
        (root->pSubtree[RIGHT] == NULL)) //nếu cây con trái và phải rỗng
    {
        // Trường hợp có 1 con - Cho nó trở thành gốc mới
        vlnode<KeyType> *ToDelete = root;
        root = root->pSubtree[(root-
            >pSubtree[RIGHT]) ? RIGHT : LEFT];
        change = HEIGHT_CHANGE;//Chiều cao cây con đã bị rút ngắn đi
    }
}

```

```

toDelete->pSubtree[LEFT] = toDelete->pSubtree[RIGHT]
= NULL;

        delete toDelete; //Xóa nút cần xóa mà con trỏ
        toDelete trỏ tới (xóa vùng nhớ đã cấp cho nút cần xóa)

        return found;      //Trả về địa chỉ nút đã xóa

    }

else

{

//Trường hợp có 2 con- ta sẽ tìm hậu bối và thay đổi vị trí của mục dữ liệu
hiện hành

root->pData = Delete(key, root->pSubtree[RIGHT],
decrease, nh_h);

}

root->Bal_fac -= decrease; // Cập nhật hệ số cân bằng
// -----
// Lập lại cân bằng nếu cần thiết- Chiều cao của nút hiện hành thay đổi
nếu một trong 2 điều sau xảy ra:

// 1) Phép quay đã làm thay đổi chiều cao của cây con
// 2) Chiều cao cây con giảm khiến cho chiều cao các cây con khác bằng
nhau, như vậy cây hiện hành có độ cân bằng zero.

// -----


change=(decrease)?((root>Bal_fac)?ReBalance(root):HEI
GHT_CHANGE):HEIGHT_NOCHANGE;

if (decrease)

{
    if (root-> Bal_fac)
    {

        change = ReBalance(root); // tái cân bằng
    }
}

```

```

        }
    else
    {
        change = HEIGHT_CHANGE; // cân bằng đã thiết lập vì
                                // chiều cao cây con giảm
    }
}
else
{
    change = HEIGHT_NOCHANGE;
}
return found;
}

```

Các code trên đây là các code khá nhỏ gọn và dễ sử dụng.

### *12. Thao tác tìm kiếm trên AVL*

// Search - Hàm tìm một khóa thỏa mãn điều kiện cho trước. Nó trả về NULL nếu không tìm thấy, ngược lại nó trả về địa chỉ của khóa tìm thấy.

```

template <class KeyType>
So_sanh<KeyType> *AvlNode<KeyType>::Search(KeyType
key, AvlNode<KeyType> * root, ss so)
{
    ss result;
    while (root && (result = root->Comp(key, so)))
    {
        root = root->pSubtree[(result < 0) ? LEFT
                                : RIGHT];
    }
    return (root) ? root->pData : NULL;
}

```

Dưới đây là hàm tạo và hủy tạo:

```

template <class KeyType>
AvlNode<KeyType>::AvlNode<So_sanh<KeyType>> *
    item) : pData(item), Bal_fac(0)
{
    Bal_fac = 0 ;
    pSubtree[LEFT] = pSubtree[RIGHT] = NULL ;
}
template <class KeyType>
AvlNode<KeyType>::~AvlNode(void)
{
    if (pSubtree[LEFT]) delete pSubtree[LEFT];
    if (pSubtree[RIGHT]) delete pSubtree[RIGHT];
}

```

Đến đây ta có thể kết thúc việc thảo luận những vấn đề cơ bản ở trên và viết nốt đoạn code cuối cùng mô phỏng lớp cây AVL trên C++

```

#include "So_sanh.h"
enum Subtr { LEFT = 0, RIGHT = 1 } ;
// AvlNode - Lớp mô phỏng cây AVL
template <class KeyType>
class AvlNode
{
public:
    // Số tối đa các nút của một cây con
    enum { MAX_SUBTREES = 2 } ;
    static Subtr
        Opposite(Subtr Tr)
    {
        return Subtree(1 - int(Tr));
    }

```

```
// Hàm tạo và hủy:
AvlNode<So_sanh<KeyType>> * item=NULL);
virtual ~AvlNode(void);

// Tạo dữ liệu của một nút
So_sanh<KeyType> *Data() const {return pData;}
// Tạo trường khóa cho nút này
KeyType Key() const { return pData->Key(); }
// -1 => Cây con trái cao hơn cây con phải
// 0 => Chiều cao cây con trái và cây con phải bằng nhau
// 1 => Cây con phải cao hơn cây con trái
short Bal_fac(void) const { return Bal_fac; }

// Tạo một mục ở đỉnh của cây con trái/phải
AvlNode *Sub(Subtree Tr) const { return
    pSubtree[Tr]; }

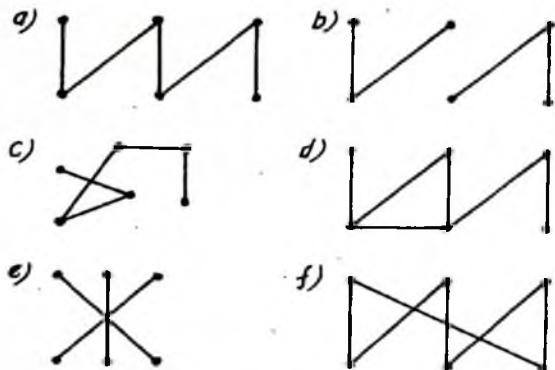
// Hàm Insert
static So_sanh<KeyType> *Search(KeyType key,
    AvlNode<KeyType> *root, ss so=bang)
static So_sanh<KeyType> *
Insert(So_sanh<KeyType> * item,
    AvlNode<KeyType> * & root)
{
    int change;
    return Insert(item, root, change);
}

// Hàm Delete
static So_sanh<KeyType> *Delete(KeyType key,
    AvlNode<KeyType> * & root, ss so=bang)
{
    int change;
    return Delete(key, root, change, cmp); }
```

```
private:  
    So_sanh<KeyType> *pData;  
    AvlNode<KeyType> *pSubtree[MAX_SUBTREES];  
    short Bal_fac;  
  
// Thực thi insertion/deletion  
    Static So_sanh<KeyType>  
        *Insert(So_sanh<KeyType> *item,  
            AvlNode<KeyType> *&root, int & change);  
    static So_sanh<KeyType> *Delete(KeyType key,  
        AvlNode<KeyType> *&root, int &change, ss so=bang);  
  
// Thực thi quay  
    static int  
        RotateOnce(AvlNode<KeyType>*&root, Subtree Tr);  
    static int RotateTwice(AvlNode<KeyType> * &  
        root, Subtree Tr);  
  
//Tái lập cân bằng  
    static int ReBalance(AvlNode<KeyType> * &  
        root);  
    ss So_sanh(KeyType key, ss cmp=bang) const;  
  
private:  
    AvlNode(const AvlNode<KeyType> &);  
    AvlNode & operator=(const  
        AvlNode<KeyType> &);  
};
```

## BÀI TẬP CHƯƠNG 5

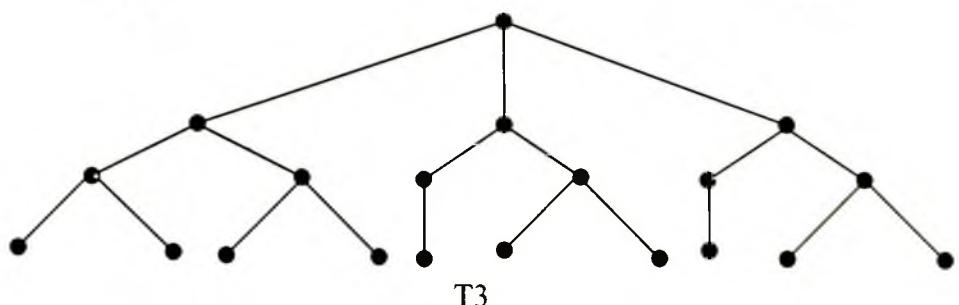
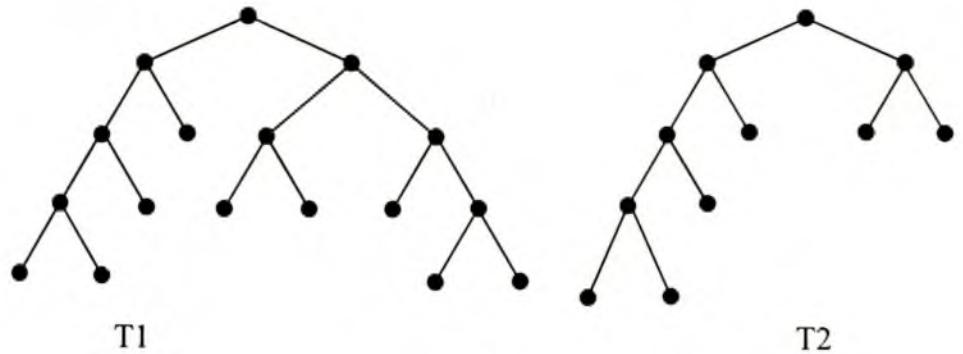
5.1. Trong các đồ thị sau, đồ thị nào là cây?



5.2. Hãy tạo BST cho bộ dữ liệu sau: mathematic, phyics, geography, zoology, métology, geology, psychology và chemistry (Dùng thứ tự Từ điển).

5.3. Vẽ cây nhị phân hoàn chỉnh có chiều cao bằng 4.

5.4. Cây m\_phân có gốc và độ cao h gọi là **cân đối** nếu tất cả các lá đều ở mức h hoặc h-1. Các cây vẽ dưới đây, cây nào là cây cân đối?



5.5. Chỉ ra rằng một đơn đồ thị  $G$  là cây nếu và chỉ nếu nó liên thông nhưng khi xoá một cạnh bất kỳ của  $G$  thì ta sẽ được một đồ thị không liên thông.

5.6. Gọi  $G$  là một đơn đồ thị có  $n$  đỉnh. Chỉ ra rằng  $G$  là cây nếu và chỉ nếu  $G$  liên thông và có  $n-1$  cạnh.

5.7. Hoặc là vẽ cây  $m$ -phân đầy đủ ( $m$ -nguyên dương) với 84 lá và có chiều cao bằng 3 hoặc chỉ ra rằng không tồn tại cây như thế.

5.8. Chứng minh phần b) và c) của định lý 4 ở mục 5.3.3

5.9. Có bao nhiêu nút trong cây hoàn chỉnh có:

a. Độ 3, chiều cao 4?

b. Độ 4 chiều cao 10?

5.10. Hai người tham gia trò chơi súc sắc chấp nhận 3 là một điểm có thể có. Trong kiểu chơi này, người Y thắng vào lần reo súc sắc đầu tiên chỉ nếu 2 hoặc 12 xuất hiện. Vẽ cây quyết định mô tả trò chơi này và tính xác suất để X thắng.

5.11. Lập công thức cho độ dài đường đi (Path) của một cây hoàn chỉnh có bậc  $d$  và độ cao  $h$ .

5.12. Trong 7 đồng xu giống nhau có một đồng giả. Đặc tính duy nhất để nhận biết đồng xu giả là trọng lượng của nó nhẹ hơn các đồng xu thật. Cách kiểm thử (Test) duy nhất để phát hiện đồng xu giả là cân các tập con phân hoạch những đồng xu. Vẽ cây quyết định mô tả quy trình thực hiện việc này.

5.13. Xây dựng lớp cây nhị phân bao gồm các thao tác cơ bản trên cây ấy.

5.14. Dùng các cấu trúc dữ liệu Queue và Tree lập trình trên C tạo mã Huffman cho một tập ký tự với xác suất xuất hiện cho trước của mỗi ký tự trong tập đó.

5.15. Xây dựng các cây nhị phân có gốc biểu diễn các biểu thức tương ứng sau:

$$a. ((x+y)/(x+3); (x+(y/3))+3; x+(y/x+3)$$

$$b. ((x+y) \uparrow 2)+((x-4)/3) \quad (\text{Ký hiệu } \uparrow \text{ biểu thị phép nâng lũy thừa})$$

$$c. ((x+2) \uparrow 3)*(y-(3+x))-5$$

5.16. Xây dựng các cây được sắp biểu diễn các mệnh đề sau:

$$a. \neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q) \quad \text{Ký hiệu } \leftrightarrow \text{biểu thị tương đương}$$

$$b. (A \cap B) - (A \cup (B-A))$$

$$c. (\neg p \wedge (q \leftrightarrow \neg p)) \vee \neg q$$

5.17. a. Hãy viết các biểu thức của bài 5.15 và dưới dạng: Prefix, Infix, Postfix.

b. Hãy viết các biểu thức của bài 5.16 dưới dạng: Prefix, Infix, Postfix.

5.18. Tính giá trị của các biểu thức tiền tố sau:

a.  $+ - * 2 3 5 / \uparrow 2 3 4$

b.  $+ - \uparrow 3 2 \uparrow 2 3 \uparrow 2 3 / 6 - 4 2$

5.19. Tính giá trị của các biểu thức hậu tố sau:

a.  $7 3 2 * - 4 \uparrow 9 3 / +$

b.  $3 2 * 2 \uparrow 5 3 - 8 4 / * -$

5.20. Hãy vẽ cây có gốc được sắp sao cho khi duyệt theo thứ tự trước ta được a, b, f, c, g, h, i, d, e, j, k, l. Trong đó a có 4 con, c có 3 con , j có 2 con , b và c đều có 1 con, còn mọi đỉnh khác đều là lá.

Khái niệm **Các công thức được tạo đúng ở dạng ký pháp tiền tố từ tập các ký tự và tập các toán tử 2 ngôi được định nghĩa để quy như sau:**

a. Nếu  $x$  là một ký tự, thì  $x$  là một công thức được tạo đúng ở dạng tiền tố.

b. Nếu  $X$  và  $Y$  là các công thức được tạo đúng và  $o$  là một toán tử thì  $oXY$  là một công thức được tạo đúng.

5.21. Trong các công thức dưới đây công thức nào là công thức được tạo đúng từ tập ký tự  $\{x, y, z\}$  và tập các toán tử 2 ngôi  $\{*, +, o\}$ :

a.  $* + + x y z$

b.  $o x y * x z$

c.  $* o x z * * x y$

d.  $* + o x x o x x x$

5.22. Chỉ ra rằng một công thức được tạo đúng bất kỳ ở dạng tiền tố từ tập các ký tự và tập các toán tử 2 ngôi có số ký tự nhiều hơn số toán tử đúng một đơn vị.

5.23. Hãy đưa ra sáu ví dụ về công thức được tạo đúng có 3 hay nhiều hơn các toán tử ở dạng hậu tố từ tập các ký tự  $\{x, y, z\}$  và tập các toán tử 2 ngôi  $\{*, +, o\}$ .

5.24. Xâu  $A \cap B - A \cap B - A$  có thể được đặt trong ngoặc đơn theo bao nhiêu cách để thu được một biểu thức trung tố (Infix).

5.25. Xâu  $\neg p \wedge q \leftrightarrow \neg p \vee \neg q$  có thể đặt trong ngoặc đơn theo bao nhiêu cách để thu được một biểu thức Infix.

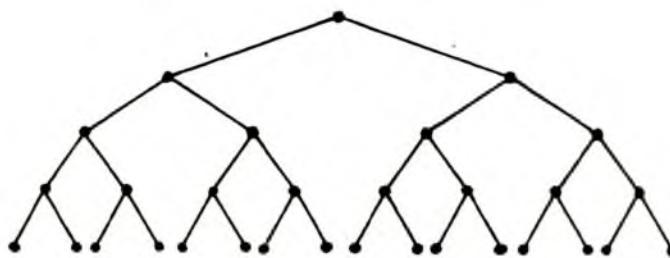
### GỢI Ý HOẶC ĐÁP ÁN

5.1. a); c); e).

5.2. Tiến trình kiến tạo BST cho bởi sơ đồ sau:

Mathematics	Mathematics Physics	Mathematics Geography Physics	Mathematics Geography Physics Zoology
	Physics > Mathematics	Geography < Mathematics	Zoology > Mathematics Zoology > Physics
Mathematics Geography Physics Meteorology Zoology Meteorology>Mathematics Meteorology<Physics	Mathematics Geography Physics Geology Zoology Geology < Mathematics Geology > Geography	Mathematics Geography Physics Geology Zoology Meteorology Psychology Psychology > Mathematics Psychology > Physics Psychology < Zoology	Mathematics Geography Physics Chemistry Zoology Chemistry < Mathematics Chemistry < Geography

5.3. Cây nhị phân hoàn chỉnh với chiều cao bằng 4:



5.6. Phần “chỉ nếu” thuộc định lý 2 và định nghĩa của cây, ta làm phần “nếu”. Giả sử  $G$  là đồ thị đơn với  $n$  đỉnh và  $n-1$  cạnh. Nếu  $G$  không là cây nó chứa (theo bài tập 5.5) một cạnh mà xóa cạnh ấy đi sẽ sinh ra đồ thị  $G'$  vẫn còn liên thông. Nếu  $G'$  không là cây thì xóa 1 cạnh của nó sẽ được  $G''$  liên thông. Lặp lại thao tác trên cho tới khi nhận được một cây. Quá trình ấy đòi hỏi nhiều nhất  $n-1$  bước vì  $G$  chỉ có  $n-1$  cạnh. Theo định lý 2 đồ thị kết quả có  $n-1$  cạnh vì nó có  $n$  đỉnh, từ đó suy ra không có cạnh nào bị xóa vì vậy  $G$  là một cây.

5.7. Theo định lý 4 không tồn tại cây như thế với  $m=2$  hoặc  $m=4$ .

5.8. a. Theo định lý 3 có  $n=m.i+1$ . Vì  $i+L=n \rightarrow L=n-i \rightarrow L=(m.i+1)-i=(m-1)i+1$ ;

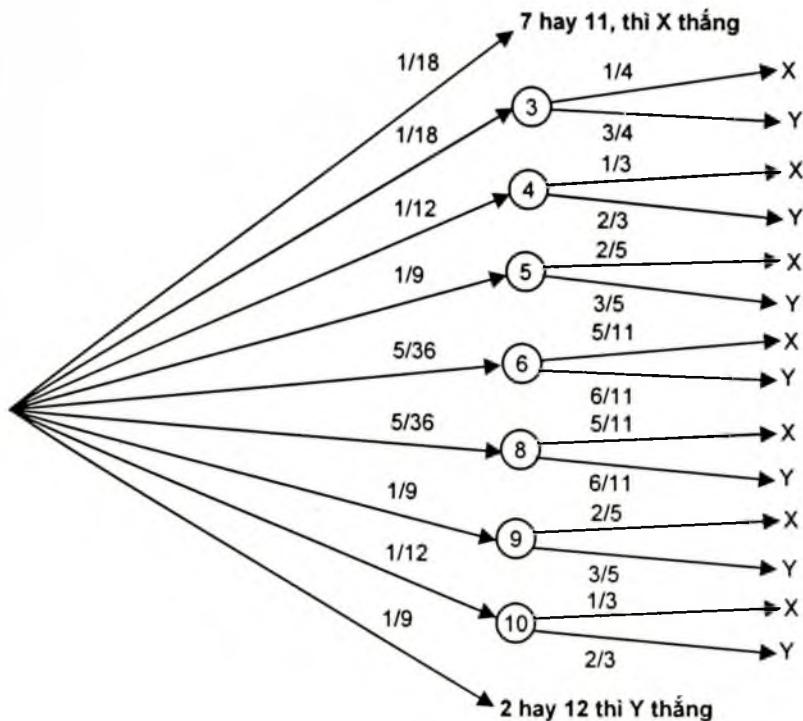
b. Ta đã có  $n=m.i+1$  và  $i+L=n$ . Vì thế  $i=n-L$ . Từ đó suy ra  $n=m(n-L)+1$ . Giải ra đối với  $n$  ta được:

$n=(mL-1)/(m-1)$ . Từ  $i=n-L$  ta nhận được:

$$\left[ \frac{mL-1}{m-1} \right] - L = \frac{L-1}{m-1}$$

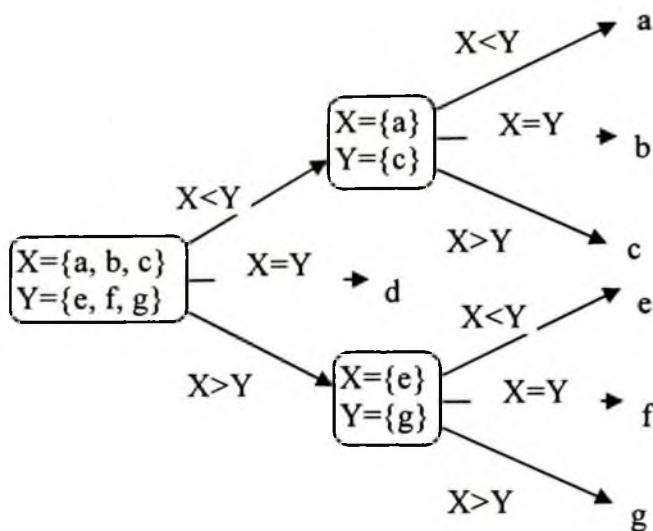
(Khi in, đặc biệt khi Photo ký tự l khó phân biệt với số 1, nên chúng ta thay l bởi L)

5.9. Dưới đây là cây quyết định mô tả trò chơi ở đầu bài:



Xác suất để X thắng trong trò chơi này là 50,68%.

5.11. Đồ thị phân hoạch tập 7 đồng xu thành các tập con để nhanh chóng phát hiện đồng xu giả có dạng sau:



5.13. Cài đặt Binary\_Class:

```

typedef enum {true, false} bool;
template <class entry>
struct Np_nut
{
    entry data;
    Np_nut <entry> *left;
    Np_nut <entry> *right;
    Np_nut ();
    Np_nut (const entry &x);
};

template <class entry>
class Np_cay //clas Np_cay: lớp cây nhị phân
{
public:
    Np_cay();
    bool empty() const;
}
  
```

```

        void dtruoc(void (*vist) (entry &)); //dtruoc:
                                            duyệt trước
        void dgiau(void (*vist) (entry &)); //điều tra
        void dsau(void (*vist) (entry &)); //điều sau
        int size() const ;
        void clear() ;
        void height() const;
        void insert(const entry &)
        Np_cay(const Np_cay <entry> &original);
        Np_cay &operator = (const Np_cay <entry>
                             &original);

~Np_cay;

protected: //các hàm đệ quy phụ trợ
void dquy_dtruoc(Np_nut<entry> *subroot,
void(*visit) (entry &))
void dquy_dgiau(Np_nut<entry> *subroot,
void(*visit) (entry &))
void dquy_dsau(Np_nut<entry> *subroot,
void(*visit) (entry &))

Np_cay<entry> *root;
};

//Khởi tạo:
template <class entry>
Np_cay<entry>::Np_cay()
{ root=NULL; }

//Kiểm tra cây có rỗng không:
template <class entry>
booll Np_cay<entry>::empty() const
{return root==NULL; }

```

```

//Các phương pháp duyệt cây:
template <class entry>
void Np_cay<entry>::dqquy_dtruoc(Np_nut<entry>
    *subroot, void(*visit) (entry &))
{
    if (subroot!=NULL)
        { (*visit) (subroot->data); //thăm gốc trước tiên
        dq_dtruoc(subroot->left, visit);
                     // rồi duyệt cây con trái
        dq_dtruoc(subroot->right, visit); //cuối cùng
                                         duyệt cây con phải
    }
}

// 2 phương pháp duyệt cây còn lại hoàn toàn tương tự, bạn đọc tự cài đặt!

//Các hàm mô tả một vài thao tác cơ bản trên cây nhị phân cài đặt bởi class:
template <class record>
class search_cay:public Np_cay<record> //tìm khóa trên cây
{ public:
    error_code: search(record &target) const;
    error_code: insert(record &target) const;
    error_code: dele(record &target) const;
    private:                                     //các hàm đệ quy phụ trợ
};

//Tìm một khóa trên cây
template <class record>
Np_nut<record>*search<record>::search_for_nut(Np_nut<record>*subroot, const record &target const
{if (subroot==NULL || subroot->data==target)
    return subroot;
else if (subroot->dat<target)  return
    search_for_nut(subroot->right, target);
}

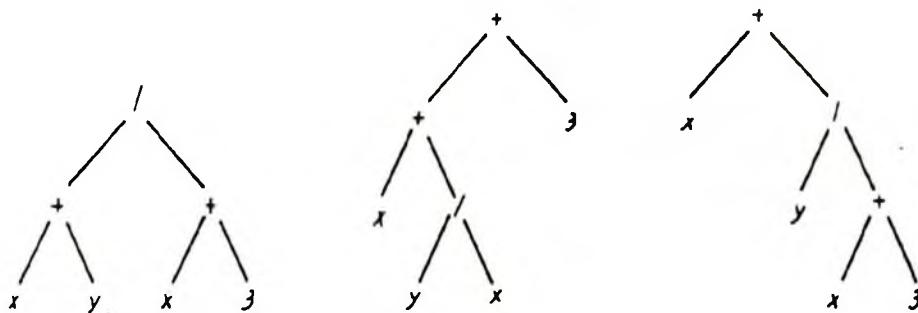
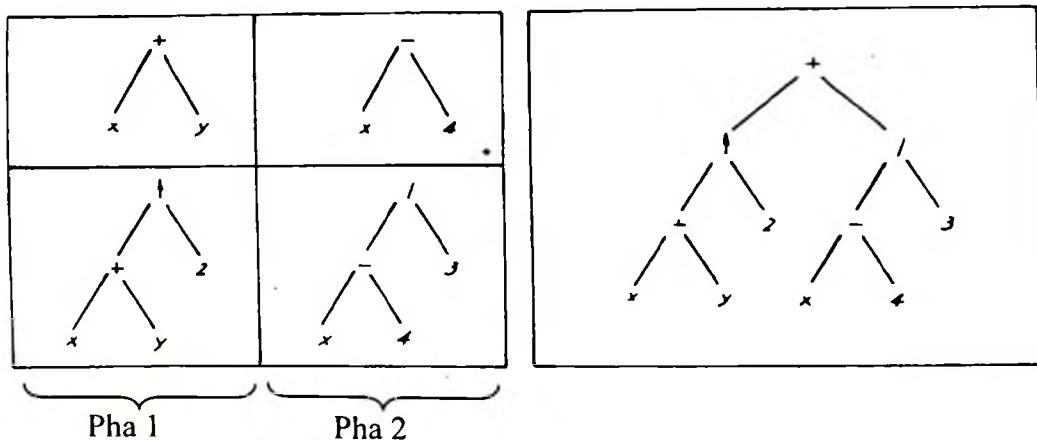
```

```
        else    return search_for_nut(subroot->left,
                                      target);
    }
```

//thao tác chèn data vào cây, trước hết phải xác định (tìm) xem trên cây đã có data cần chèn chưa? Nếu đã có thì không chèn được, ngược lại thì chèn. Đầu tiên xây dựng hàm tìm:

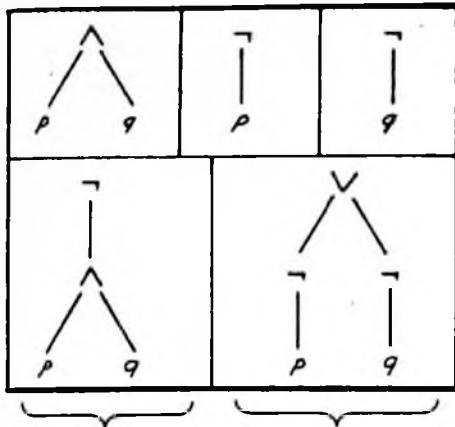
```
error search<record>::insert(const record &newdata);
template <class record>
error_code search<record>::insert(const &newdata)
{
    return search_and_insert(root, newdata);
}
template <class record>
error_code
search<record>::search_and_insert(Np_nut<record>
                                   *&subroot, const record &newdata)
{
    if (subroot==NULL)
    { subroot=new Np_nut<record> (newdata);
      return success;
    }
    else if (newdata<subroot->data)
        return search_and_insert(subroot->left,
                                  newdata);
    else if (newdata>subroot->data)
        return search_and_insert(subroot->right,
                                  newdata);
    else return duplicate_error; //data cần chèn đã tồn tại rồi!
}
//Thao tác xóa làm tương tự (cần lưu ý: nút cần xóa có một con và nút cần xóa có 2 con)
```

5.15. a.

b.  $((x+y) \uparrow 2) + (x-4)/3:$ 

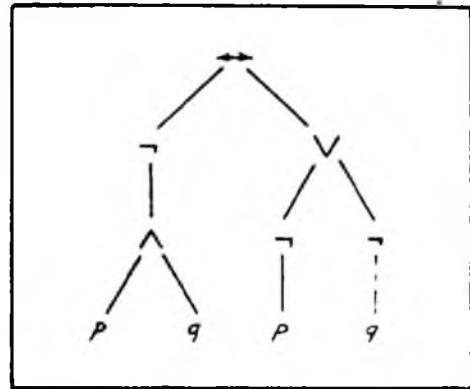
**Lý giải:** Phương pháp quen thuộc để vẽ CNP cho biểu thức đã nêu là xây dựng từ dưới lên: Đầu tiên vẽ cây con cho biểu thức  $x+y$ . Tiếp theo kết hợp với cây con lớn hơn  $(x+y) \uparrow 2$ . Pha thứ hai của công việc này là vẽ cây con  $(x-4)$  tiếp theo kết hợp thành cây con  $(x-4)/3$ . Cuối cùng kết hợp 2 cây con biểu diễn  $(x+y) \uparrow 2$  và cây biểu diễn  $(x-4)/3$  để thu được cây có gốc được sắp của biểu thức đã cho.

5.16. a.  $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$ 



Pha 3

Pha 4



**Lý giải:** Cách làm cũng như bài 5.15 b). Trước tiên ta tạo cây con  $\neg p$  và  $\neg q$  (trong đó  $\neg$  là toán tử phủ định). Tương tự ta vẽ cây con  $p \wedge q$ . Tiếp theo là các cây con  $\neg(p \wedge q)$  và  $(\neg p) \vee (\neg q)$ . Cuối cùng kết hợp hai cây con ở pha 1 và pha 2 để được cây có gốc được sắp cần tìm.

5.17 a. Có một cách *nhanh nhất* và *thật đơn giản* là duyệt cây nhị phân biểu diễn  $((x+y) \uparrow 2) + (x-4)/3$  ở đáp án 5.15.b. theo thứ tự trước (PreOrder) ta có ngay:  $+ \uparrow + x \ y \ 2 \ / - \ x \ 4 \ 3$ . Hai dạng còn lại cũng làm nhờ cách duyệt cây theo thứ tự giữa và theo thứ tự sau.

b. Duyệt cây tìm được ở đáp án 5.16.a. theo thứ tự trước, thứ tự sau rồi thứ tự giữa ta được các kết quả tương ứng:

\*Prefix:  $\leftrightarrow \neg \wedge p q \vee \neg p \neg q;$

\*Postfix :  $p q \wedge \neg p \neg q \neg \vee \leftrightarrow ;$

\*Infix:  $(\neg(p \wedge q)) \leftrightarrow ((\neg p) \vee (\neg q))$

Các dạng Prefix và Postfix là rõ ràng, không nhập nhằng nước đôi và vì có thể dễ dàng tính giá trị của nó mà không phải rà tới rà lui, nên chúng được sử dụng rất nhiều trong tin học. Các biểu diễn này đặc biệt có lợi khi xây dựng các translator (các bộ dịch). Làm tương tự, chúng ta có kết quả của 5.17.b:

$\neg \cap A \cup A = AB ; AB \cap ABA = \cup ; ((A \cap B) - (A \cup (B - A)))$

5.18. a. Quy trình tính giá trị của biểu thức *tiền tố* này là từ *phải sang trái* và làm các phép toán với 2 *toán hạng* ở *bên phải* nó:

$$+ - * 2 3 5 / \uparrow 2 3 4$$

$\underbrace{2}_{2} \uparrow \underbrace{3}_{3}=8$

$$+ - * 2 3 5 / 8 4$$

$\underbrace{8}_{8}/\underbrace{4}_{4}=2$

$$+ - \underbrace{2 3 5}_{2}$$

$2*3=6$

$$+ - \underbrace{6 5}_{6} 2$$

$6-5=1$

$$\underbrace{+ 1 2}_{1}$$

$1+2=3 \rightarrow$  Giá trị biểu thức đã cho là: 3

5.19.a. Các bước tính giá trị của biểu thức hậu tố này tiến hành từ trái qua phải và thực hiện các phép toán với hai toán hạng ở bên trái các toán tử:

$$7 \underbrace{2 3 * - 4}_{2*3=6} \uparrow 9 3 / +$$

$$\underbrace{7 6 - 4}_{7-6=1} \uparrow 9 3 / +$$

$$\underbrace{1 4 \uparrow 9 3}_{1\uparrow 4=1} / +$$

$$1 \underbrace{9 3}_{9/3=3} / +$$

$$\underbrace{1 3}_{1+3=4} +$$

5.22. Chứng minh bằng quy nạp. Gọi  $S(X)$  và  $O(X)$  lần lượt là số các ký tự và số các toán tử trong công thức được tạo đúng X. Mệnh đề là đúng với công thức được tạo đúng với độ dài là 1. Giá sử mệnh đề là đúng với mọi công thức được tạo đúng với độ dài nhỏ hơn n. Công thức được tạo đúng với độ dài n phải có dạng  $*XY$ , trong đó  $*$  là toán tử và X, Y là các công thức được tạo đúng với độ dài nhỏ hơn n. Khi ấy theo giả thiết quy nạp ta có:  $S(*XY)=S(X) + S(Y) = (O(X)+1) = (O(Y)+1)=O(X) + O(Y) + 2$ .

Vì  $O(*XY) = 1 + O(X) + O(Y)$  nên suy ra:  $S(*XY) = O(*XY) + 1$ . ĐPCM!

5.23. Dưới đây là 6 ví dụ có thể có:

$xy + zxo + xo$ ;  $xyz++yx++$ ;  $xyxyoooxyzo+$ ;  $xzxzz+o$ ;  $yyyyoooo$ ;  $zx+yz+o$

5.24. Có tất cả 14 cách.

## **PHỤ LỤC**

### **LỚP NP - ĐẦY ĐỦ**

Trong lý thuyết độ phức tạp tính toán, lớp **NP-đầy đủ** là một lớp các bài toán quyết định. Một bài toán  $L$  là **NP-đầy đủ** nếu nó nằm trong lớp NP (lời giải cho  $L$  có thể được kiểm chứng trong thời gian đa thức) và là NP - khó (mỗi bài toán trong NP đều có thể quy về  $L$  trong thời gian đa thức).

Mặc dù bất kỳ lời giải nào cho mỗi bài toán đều có thể được kiểm chứng nhanh chóng, song hiện chưa có cách nào tìm ra được lời giải đó một cách hiệu quả. Thời gian thực thi của tất cả các thuật toán hiện tại cho những bài toán này đều tăng rất nhanh theo kích thước dữ liệu vào của bài toán. Vì vậy ngay cả những trường hợp có kích thước tương đối lớn của dữ liệu vào đòi hỏi thời gian hàng tỷ năm để giải. Do đó, việc xác định xem những bài toán này có thể được giải quyết nhanh chóng hay không (thường gọi là bài toán P so với NP) là một trong những bài toán mở của khoa học máy tính hiện nay.

Các bài toán NP-đầy đủ xuất hiện thường xuyên trong thực tế nên mặc dù chưa có giải thuật trong thời gian đa thức cho chúng, các nhà nghiên cứu vẫn tìm cách giải quyết chúng thông qua các phương pháp khác như thuật toán xấp xỉ, nhân tử hóa, v.v...

#### *Tổng quan*

Lớp **NP-đầy đủ** là tập hợp các bài toán NP-khó trong NP.

Lớp **NP-đầy đủ** được quan tâm nghiên cứu bởi khả năng kiểm chứng nhanh chóng lời giải (NP) dường như có liên hệ với khả năng tìm kiếm nhanh chóng lời giải (P). Hiện vẫn chưa biết được nếu mọi bài toán trong NP đều có thể được giải quyết nhanh chóng hay không (đây chính là bài toán P so với NP). Tuy nhiên, nếu bất kỳ một bài toán nào trong NP-đầy đủ có thể được giải quyết nhanh chóng, thì theo định nghĩa của NP-đầy đủ, mọi bài toán trong NP đều có thể được giải quyết nhanh chóng.

#### *Lịch sử*

Khái niệm NP-đầy đủ được đưa ra bởi Stephen Cook năm 1971 trong bài báo mang tên "The complexity of theorem-proving procedures". Tuy nhiên tên gọi NP-đầy đủ không xuất hiện trong bài báo này mà được đưa ra sau đó. Trong

đó Cook đã chứng minh định lý Cook-Levin (Leonid Levin cũng chứng minh định lý này một cách độc lập cùng thời gian với Cook). Định lý này chứng minh bài toán Circuit-SAT là NP-dài dù. Năm 1972, Richard Karp chứng minh 21 bài toán khác cũng là NP-dài dù. Từ sau đó đến nay, hàng nghìn bài toán đã được chứng minh là NP-dài dù. Nhiều bài toán quan trọng trong số đó được liệt kê trong cuốn "Computers and Intractability: A Guide to the Theory of NP-Completeness" của Garey và Johnson.

## TÀI LIỆU THAM KHẢO

1. Mark Allen Weiss. Florida International University. **Data Structs & Algorithm Anaysis In C++**. Second Edition. Addison-Wesley. An Imprint of Addison Wesley Longman, Inc. 1999.
2. Frank M. Carrano, University of Rhode Island & Janet J. Prichard, Bryant College. **Data Abstraction And Problem Solving With C++**. Walls And Mirrors. Third Edition. Addition-Wesley. 2000.
3. Robert Sedghewick and Kevin Wayne, Princeton University (USA). **Algorithms. 4th Edition**. Addition-Wesley Publishing Co. 2012.
4. A.V Aho, J.E Hopcroft, J.D Ullman. **Data structures and Algorithm**. Addition-Wesley.1983
5. Knut D. **The Art of Programming**. Tom 1, 2, 3. Addition-Wesley. 1971
6. Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest. **Introduction to Algorithms**. MIT Press. 1998.
7. Frank L. Friedman & Elliot B.Koffoman, Temple University. **Problem Solving, Abstraction and Design Using C++**. Second Edition. Addition-Wesley. 1996.
8. Kenneth H.Rosen. **Discrete Mathematics and Its Applications**. McGraw-Hill. 1994.
9. John R.Hubbard, Ph.D. Professoe of University of Richmond. **455 bài tập Cấu trúc Dữ liệu Cài đặt bằng C++** (nguyên bản tiếng Anh tái bản lần thứ 8). Biên dịch: Minh Trung, Gia Việt. NXB Thông Kê. 2003
10. Trần Hạnh Nhi, Dương Anh Đức. **Giáo trình Cấu trúc Dữ liệu và Giải thuật**. NXB ĐHQG TP Hồ Chí Minh. 2009.
11. Nguyễn Xuân Huy. **Sáng tạo trong Thuật toán và Lập trình**. T1, 2, 3. Nhà Xuất bản Thông tin và Truyền thông. 2011.
12. Engelhardt M. **A Group-bassed Search for Solutions of N-Queens Problem**. 2007. Discrete Mathematics. Vol 307. pp 2535-2551.

**13. The Knight's Tour by Mark R. Keen on Website:**  
<http://www.markkeen.com/knight.html>

**14. "The Problem of the Knight: A Fast and Simple Algorithm" by Arnd Roth on Website: [http://sunny.mpimf-heidelberg.mpg.de/...](http://sunny.mpimf-heidelberg.mpg.de/)**

**15. Đinh Mạnh Tường. Cấu trúc Dữ liệu & Thuật toán. NXB Khoa Học và Kỹ Thuật. 2000.**

**16. Các Website: TopCoder.Com; BookBoon.Com**

**17. Một vài bài thực hành tốt về Cấu trúc dữ liệu của đồng nghiệp:**  
Tiến sĩ Phan Đăng Cầu. HV. Công Nghệ - Bưu Chính - Viễn Thông Hà Nội.  
(Tài liệu lưu hành nội bộ). 2008.

# MỤC LỤC

<i>Lời nhà xuất bản</i> .....	3
<i>Lời nói đầu</i> .....	5
<b>Chương 1. CÁC KHÁI NIỆM CHUNG VỀ THUẬT TOÁN VÀ CẤU TRÚC DỮ LIỆU</b> .....	7
<b>1.1. Một vài khái niệm về Thuật toán</b> .....	7
1.1.1. Định nghĩa thuật toán.....	7
1.1.2. Các tính chất của thuật toán.....	7
1.1.3. Ba đặc trưng của thuật toán .....	9
1.1.4. Các cách diễn đạt thuật toán (Expressing Algorithms).....	9
1.1.5. Đánh giá thời gian thực hiện (độ phức tạp) của thuật toán.....	13
1.1.6. Khái niệm ô lớn (Big_O, ký hiệu là O()).....	14
1.1.7. Hai luật cơ bản của Big_O .....	16
1.1.8. Đánh giá ĐPT của các câu lệnh cơ bản của C/C <sup>++</sup> .....	18
1.1.9. Đánh giá ĐPT của một số chương trình (thuật toán) cơ bản. ....	21
<b>1.2. Kiểu dữ liệu và Cấu trúc dữ liệu</b> .....	23
1.2.1. Kiểu dữ liệu là gì?.....	23
1.2.2. Định nghĩa cấu trúc dữ liệu.....	24
1.2.3. Các kiểu dữ liệu và cấu trúc dữ liệu trong C/C <sup>++</sup> .....	24
1.2.4. Vai trò của cấu trúc dữ liệu trong một chương trình ứng dụng tin học .....	33
<i>Bài tập chương 1</i> .....	38
<i>Gợi ý hoặc đáp án</i> .....	43
<b>Chương 2. TÌM KIẾM TRONG VÀ SẮP XẾP TRONG</b> .....	65
<b>A. Các thuật toán tìm kiếm</b> .....	65
<b>2.1. Tìm kiếm tuần tự (Sequential Search)</b> .....	65
2.1.1. Ý tưởng của thuật toán.....	65

2.1.2. Các bước của thuật toán.....	65
2.1.3. Cài đặt.....	66
2.1.4. Độ phức tạp (ĐPT) của tìm kiếm tuần tự .....	68
<b>2.2. Tìm kiếm nhị phân (Binary Search) .....</b>	<b>68</b>
2.2.1. Những lưu ý đầu tiên về thuật toán này .....	68
2.2.2. Ý tưởng của thuật toán.....	68
2.2.3. Các bước của thuật toán.....	70
2.2.4. Cài đặt .....	70
2.2.5. Độ phức tạp của Binary Search .....	71
<b>B. Các thuật toán sắp xếp.....</b>	<b>72</b>
<b>2.3. Mở đầu .....</b>	<b>72</b>
<b>2.4. Liệt kê các thuật toán sắp xếp .....</b>	<b>72</b>
2.4.1. Các thuật toán sắp xếp đơn giản, thông dụng .....	72
2.4.2. Các thuật toán khó (khi cài đặt) và ít quen thuộc đối với học sinh, sinh viên .....	72
<b>2.5. Sắp xếp lựa chọn (Selection Sort).....</b>	<b>73</b>
2.5.1. Ý tưởng của thuật toán.....	73
2.5.2. Nghịch thế.....	73
2.5.3. Sơ đồ mô tả cơ chế hoạt động Selection Sort (còn gọi là sơ đồ “dò vết”).....	73
2.5.4. Các bước thực thi Selection Sort .....	74
2.5.5. Hàm diễn đạt thuật toán Selection Sort .....	75
2.5.6. Cài đặt Selection Sort .....	75
2.5.7. Độ phức tạp của Selection Sort.....	77
<b>2.6. Sắp xếp chèn (Insert Sort).....</b>	<b>77</b>
2.6.1. Những nhận xét và quan niệm ban đầu.....	77
2.6.2. Ý tưởng của thuật toán.....	77
2.6.3. Sơ đồ mô tả cơ chế làm việc của Insert Sort (sơ đồ dò vết) .....	77
2.6.4. Các bước thực thi Insert Sort .....	78

2.6.5. Hàm diễn đạt thuật toán Insert Sort .....	79
2.6.6. Cài đặt Insert Sort .....	79
2.6.7. Độ phức tạp của Insert Sort .....	81
<b>2.7. Sắp xếp đổi chỗ (Interchange Sort).....</b>	<b>81</b>
2.7.1. Mở đầu.....	81
2.7.2. Ý tưởng .....	81
2.7.3. Sơ đồ mô tả cơ chế hoạt động của Interchange Sort (sơ đồ dò vết) .....	82
2.7.4. Các bước thực thi Interchange Sort .....	83
2.7.5. Hàm diễn đạt thuật toán Interchange Sort .....	83
2.7.6. Độ phức tạp của Interchange Sort.....	83
<b>2.8. Sắp xếp nồi bọt (Bubble Sort).....</b>	<b>84</b>
2.8.1. Xuất xứ của cái tên “Nồi bọt” (Bubble) .....	84
2.8.2. Ý tưởng của thuật toán.....	84
2.8.3. Các bước thực thi Bubble Sort.....	84
2.8.4. Sơ đồ dò vết của Bubble Sort (tương tự các trường hợp trên, độc giả tự làm).....	84
2.8.5. Hàm diễn đạt thuật toán Bubble Sort.....	85
2.8.6. Độ phức tạp của thuật toán .....	85
<b>2.9. Sắp xếp rung (Shake Sort) .....</b>	<b>85</b>
2.9.1. Ý tưởng của thuật toán.....	85
2.9.2. Các bước thực thi Shake Sort .....	85
2.9.3. Sơ đồ dò vết (độc giả tự làm).....	86
2.9.4. Hàm diễn đạt thuật toán Shake Sort.....	86
2.9.5. Độ phức tạp của thuật toán Shake Sort.....	87
<b>2.10. Sắp xếp với độ dài bước giảm dần</b>	
- Còn gọi là <b>sắp xếp theo cách nã pháo (Shell Sort)</b> .....	87
2.10.1. Ý tưởng của thuật toán Shell Sort.....	87
2.10.2. Các bước thực thi Shell Sort.....	87
2.10.3. Sơ đồ dò vết (độc giả tự làm).....	88

2.10.4. Hàm diễn đạt thuật toán Shell Sort .....	88
2.10.5. Cài đặt Shell Sort.....	88
2.10.6. Độ phức tạp của thuật toán Shell Sort .....	90
<b>2.11. Sắp xếp nhanh (Quick Sort) .....</b>	<b>90</b>
2.11.1. Ý tưởng của thuật toán.....	90
2.11.2. Các bước thực thi Quick Sort .....	91
2.11.3. Bảng dò vết.....	92
2.11.4. Hàm diễn đạt thuật toán Quick Sort.....	92
2.11.5. Cài đặt Quick Sort .....	93
2.11.6. Độ phức tạp của thuật toán .....	94
<b>2.12. Sắp xếp vun đong (Heap Sort).....</b>	<b>95</b>
2.12.1. Định nghĩa Heap .....	95
2.12.2. Các tính chất của Heap .....	95
2.12.3. Các bước thực thi Heap Sort.....	96
2.12.4. Xây dựng hàm hiệu chỉnh dãy ban đầu thành Heap .....	96
2.12.5. Xây dựng tạo hàm Heap .....	97
2.12.6. Hàm thực thi Heap Sort .....	97
2.12.7. Cài đặt Heap Sort.....	97
2.12.8. Bảng dò vết Heap Sort.....	100
2.12.9. Độ phức tạp của Heap Sort .....	101
<b>2.13. Sắp xếp trộn (Merge Sort) .....</b>	<b>101</b>
2.13.1. Ý tưởng .....	101
2.13.2. Các bước thực thi Merge Sort.....	101
2.13.5. Xây dựng hàm trộn các Run .....	102
2.13.6. Xây dựng hàm xếp trộn .....	104
2.13.7. Cài đặt Merge Sort.....	104
2.13.8. Độ phức tạp của Merge Sort .....	107
<b>2.14. Sắp xếp cơ sở (Radix Sort hay Post Sort) .....</b>	<b>107</b>
2.14.1. Mở đầu.....	107

2.14.2. Ý tưởng .....	108
2.14.3. Cơ chế làm việc của Post Sort .....	108
2.14.4. Các bước thực thi thuật toán Post Sort .....	111
2.14.5. Cài đặt Post Sort .....	111
2.14.6. Hàm phân loại các phần tử .....	111
<i>Bài tập chương 2 .....</i>	112
<i>Gợi ý hoặc đáp án.....</i>	114
<b>Chương 3. MỘT SỐ CHIẾN LƯỢC THIẾT KẾ THUẬT TOÁN .....</b>	<b>125</b>
<b>3.1. Chiến lược chia đế (dế) trị.....</b>	<b>125</b>
3.1.1. Ý tưởng chủ đạo của chiến lược này .....	125
3.1.2. Một số minh họa cho chiến lược “chia đế (dế) trị” .....	125
<b>3.2. Chiến lược đế quy .....</b>	<b>127</b>
3.2.1. Đế quy là gì?.....	127
3.2.2. Đế quy trong khoa học máy tính.....	127
3.2.3. So sánh đế quy và lặp .....	129
3.2.4. Nguyên tắc thiết kế đế quy .....	131
3.2.5. Các loại đế quy .....	131
<b>3.3. Đế quy và quay lui (Recursion and Backtracking).....</b>	<b>140</b>
3.3.1. Đặc trưng chủ yếu của chiến lược quay lui .....	140
3.3.2. Minh họa kinh điển cho thuật toán quay lui: bài toán Tám quân hậu.....	141
3.3.3. Tổng quát hóa bài toán tám hậu.....	144
3.3.4. Bài toán Mã đi tuần (Ví dụ kinh điển 2 minh họa cho chiến lược quay lui) .....	146
<b>3.4. Thuật toán Tham ăn (Greedy Algorithm - GA) .....</b>	<b>15</b>
3.4.1. Các thành phần (components) và hai đặc trưng của GA .....	15
3.4.2. Một ứng dụng của GA cho một bài toán cụ thể .....	15
<b>3.5. Thuật toán quy hoạch động .....</b>	<b>15</b>
3.5.1. Mở đầu .....	15

3.5.2. Nhận diện các bài toán không tối ưu hóa có thể giải được bằng QHĐ .....	158
3.5.3. Dùng QHĐ cho các bài toán tối ưu hóa.....	160
<b>3.6. Thuật toán nhánh và cận .....</b>	<b>164</b>
3.6.1. Ý tưởng chủ đạo của chiến lược nhánh và cận .....	164
3.6.2. Tóm tắt bản chất của chiến lược BB.....	164
3.6.3. Mã giả tổng quát diễn đạt thuật toán.....	165
<i>Bài tập chương 3 .....</i>	168
<i>Gợi ý hoặc đáp án.....</i>	172
<b>Chương 4. CÁC KIỀU DỮ LIỆU TRÙU TƯỢNG VÀ BIẾN NHÓ ĐỘNG.....</b>	<b>178</b>
<b>4.1. Các kiến thức cơ bản về kiểu dữ liệu trùu tượng (kiểu dữ liệu động) .....</b>	<b>178</b>
4.1.1. Đặt vấn đề .....	178
4.1.2. Định nghĩa kiểu dữ liệu trùu tượng .....	179
4.1.3. Sự khác nhau giữa ADTs với cấu trúc dữ liệu.....	180
4.1.4. Kiểu dữ liệu con trỏ và biến thuộc kiểu con trỏ.....	180
<b>4.2. Danh sách tuyến tính trùu tượng .....</b>	<b>183</b>
<b>4.3. Danh sách liên kết trùu tượng .....</b>	<b>184</b>
4.3.1. Định nghĩa DSLKD .....	184
4.3.2. Các thao tác cơ bản trên DSLKD .....	185
4.3.3. Các phương án Sắp xếp DSLKD .....	195
4.3.4. Một số thuật toán sắp xếp dữ liệu của DSLKD .....	197
4.3.5. Cài đặt hoàn chỉnh DSLKD cho một áp dụng thực tế .....	205
<b>4.4. Các danh sách liên kết đơn đặc biệt.....</b>	<b>213</b>
4.4.1. Kiểu dữ liệu trùu tượng ngăn xếp .....	213
4.4.2. Kiểu dữ liệu trùu tượng Hàng đợi (Queue ADT) .....	226
4.4.3. Kiểu dữ liệu trùu tượng Hàng đợi ưu tiên (Priority Queues ADT - PQ).....	235

<b>4.5. Kiểu dữ liệu trùu tượng danh sách liên kết đôi (Double Linked List ADT) .....</b>	239
4.5.1. Định nghĩa .....	239
4.5.2. Các thao tác cơ bản trên DSLK_Đôi .....	240
<b>4.6. Kiểu dữ liệu trùu tượng danh sách liên kết vòng .....</b>	248
4.6.1. Định nghĩa danh sách liên kết vòng - DSLKV (Circular Linked List - CLL-).....	248
4.6.2. Các thao tác cơ bản trên DSLKV .....	249
<i>Bài tập chương 4 .....</i>	252
<i>Gợi ý hoặc đáp án.....</i>	257
<b>Chương 5. CÁU TRÚC CÂY (TREE STRUCTURE).....</b>	285
<b>5.1. Các khái niệm cơ bản về cây.....</b>	285
5.1.1. Định nghĩa đệ quy về cây .....	285
5.1.2. Định nghĩa phi đệ quy về cây .....	286
5.1.3. Cây và những khái niệm liên quan .....	286
5.1.4. Cây được sắp .....	287
5.1.5. Nút con trưởng và nút anh liền kề .....	288
5.1.6. Cây gán nhãn .....	288
5.1.7. Rừng .....	288
5.1.8. Định lý 1 .....	288
<b>5.2. Định nghĩa cây m_phân .....</b>	289
<b>5.3. Các tính chất của cây m_phân.....</b>	289
5.3.1. Định lý 2 .....	289
5.3.2. Định lý 3 .....	289
5.3.3. Định lý 4 .....	289
5.3.4. Định lý 5 .....	290
5.3.5. Hệ quả .....	290
<b>5.4. Cây nhị phân - CNP.....</b>	291
5.4.1. Định nghĩa phi đệ quy về cây nhị phân (Binary tree - BT) .....	291

5.4.2. Định nghĩa đệ quy về cây nhị phân.....	291
5.4.3. Khai báo cấu trúc CNP trong ngôn ngữ C .....	292
5.4.4. Một ứng dụng của CNP .....	292
5.4.5. Các thao tác (Operations) trên CNP .....	293
<b>5.5. Cây nhị phân tìm kiếm (Binary Search Tree - BST) .....</b>	<b>293</b>
5.5.1. Mở đầu .....	293
5.5.2. Định nghĩa đệ quy cây BST.....	294
5.5.3. Định nghĩa phi đệ quy cây BST.....	294
5.5.4. Các thao tác trên cây BST.....	295
5.5.5. Cài đặt hoàn chỉnh Cây nhị phân tìm kiếm.....	301
5.5.6. Độ phức tạp của các thao tác trên BST .....	307
<b>5.6. Cây nhị phân cân bằng (Cây NPCB) .....</b>	<b>307</b>
5.6.1. Cây NPCB hoàn toàn (cây NPCBHT).....	307
5.6.2. Cây nhị phân cân bằng (Cây NPCB, tên khác: cây AVL) .....	308
<i>Bài tập chương 5 .....</i>	335
<i>Gợi ý hoặc đáp án.....</i>	338
<i>Phụ lục: Lớp NP - đầy đủ .....</i>	349
<i>Tài liệu tham khảo .....</i>	351

# CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

(PHÂN TÍCH VÀ CÀI ĐẶT TRÊN C/C")

(Tập 1)

---

**Chịu trách nhiệm xuất bản**

*Giám đốc - Tổng biên tập*

**NGUYỄN THỊ THU HÀ**

**Biên tập:**

**NGÔ MỸ HẠNH  
NGUYỄN LONG BIÊN  
TRƯƠNG MINH ĐỨC**

**Trình bày sách:**

**NGUYỄN ĐỨC LỘC**

**Sửa bản in:**

**TRƯƠNG MINH ĐỨC**

**Trình bày bìa:**

**TRẦN HỒNG MINH**

---

## NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

**Trụ sở:** Số 9, Ngõ 90, Phố Ngụy Như Kon Tum, Quận Thanh Xuân, TP. Hà Nội

ĐT Biên tập: 04.35772143

ĐT Phát hành: 04.35772138

E-mail: nxb.tttt@mic.gov.vn

Fax: 04.35579858

Website: [www.nxbthongtintruyenthong.vn](http://www.nxbthongtintruyenthong.vn)

**Chi nhánh TP. Hồ Chí Minh:** 8A đường D2, P25, Quận Bình Thạnh, TP. Hồ Chí Minh

Điện thoại: 08.35127750, 08.35127751

Fax: 08.35127751

E-mail: cnsq.nxbtttt@mic.gov.vn

**Chi nhánh TP. Đà Nẵng:** 42 Trần Quốc Toản, Quận Hải Châu, TP. Đà Nẵng

Điện thoại: 0511.3897467

Fax: 0511.3843359

E-mail: cndn.nxbtttt@mic.gov.vn

---

In 700 bản, khổ 16x24cm, tại Công ty In Hải Nam

Số đăng ký kế hoạch xuất bản: 172-2014/CXB/4-67/TTTT

Số quyết định xuất bản: 163/QĐ-NXB TTTT ngày 06 tháng 6 năm 2014

In xong và nộp lưu chiểu tháng 6 năm 2014

